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