Building Secure Applications with Cryptography in .NET

Course Overview

Course Overview

Hi, everyone. My name is Stephen Haunts, and welcome to my course, Building Secure Applications with Cryptography in .NET. I am a freelance software developer, trainer, and writer, and one of my specialisms is enterprise software security. As a software developer working in an organization, you have a big responsibility to your employers and their customers to protect their data to make it safer against data breaches. Cryptography has a reputation as being very complicated, but in practice it's not that hard to implement with some guidance. In this course, I'm going to take you through all of the main cryptographic primitives available in the .NET Framework, .NET Core, .NET 5, and beyond. We're going to cover some of the following topics: secure random number generation, hashing and password-based key derivation functions, symmetric encryption with AES, including the recently added GCM mode, asymmetric encryption with RSA, and creating digital signatures of your data. Towards the end of this course, I'm going to show you how to combine lots of these different primitives together to perform what is called hybrid encryption, where you can take advantage of all the best features of each tool. By the end of this course, you'll have a firm grasp with the theory and practice when it comes to encrypting sensitive data in your enterprise systems. Cryptography doesn't have to feel complicated or scary. I hope that I can demystify this subject with of lots of demos and included code samples and give you all the tools you need to become an expert in no time. Before beginning this course, you should be reasonably familiar with C# and your IDE of choice from Visual Studio for Windows or Mac, Visual Studio Code, or JetBrains Rider. The majority of what I teach in this course is completely cross-platform between Windows, macOS, and Linux, apart from a few small sections, which I'll call out during this course. I hope you'll join me in this journey to learn about cryptography in the .NET platform with this course. Building Secure Applications with Cryptography in .NET, here at Pluralsight.

Introduction

Introduction

Hi, my name is Stephen Haunts from Pluralsight, and welcome to my course, Building Secure Applications with Cryptography in .NET. This course is aimed at .NET developers, that is, developers who might still be using a more traditional .NET framework and also developers using .NET Core, .NET 5, and any future versions of the platform. You don't need to be a security expert to use this course. The only requirement is that you are reasonably familiar with C#, and you can navigate yourself around your IDE of choice, such as Visual Studio for Windows or Mac or JetBrains Rider. This course is aimed at those developers who work in environments where protecting and securing the data of your company is of the utmost importance. This could be protecting personally identifiable information of your customers, securing credit card information for payment transactions, or protecting any sensitive data that has commercial value to your organization. In this course we'll cover the practical elements of cryptography using the .NET Framework, and more specifically, the system.security.cryptography namespace. We'll start the course by looking at cryptographic random numbers. The ability to generate cryptographically secure random numbers is essential to effective cryptography. What you learn in this section of the course will be used throughout the remainder of the course. After looking at random numbers, we'll then take a detailed look at hashing. We'll first look at the theory and then dive into how to do hashing. Hashing is commonly used to check the integrity of data and provide an equivalent to a digital fingerprint. Once we've looked at hashing, we'll then go a step further and discuss how to securely store passwords for your systems. We'll look at the common pitfalls and how easy they are to compromise, and then look at a couple of recognized and secure password storage mechanisms. We'll look at the built-in password-based key derivation functions in the .NET Framework. After we've looked at some secure password storage, we'll then look at symmetric encryption in .NET. This is the area of cryptography that people generally think of when they talk about cryptography. In this module we'll look at the advanced encryption standard. We'll also touch on the complexities of securely exchanging encryption keys between a sender and their recipient. By the end of this module, you'll have a set of classes that you can use straightaway in your applications. After looking at symmetric encryption, we'll take a look at asymmetric encryption. Here we'll look at an encryption scheme that makes it easier to share keys. We'll specifically, look at the RSA encryption algorithm. By the end of this module, you'll have a working set of classes that you can use in your application to enable you to leverage to full use of the RSA system. After we have looked at RSA and asymmetric cryptography, we'll then take a look at digital signatures. A digital signature is a technique used to validate the authenticity and integrity of a message, software, or digital document. We'll look at a bit of theory behind digital signatures and then look at their practical usage within .NET. Once we've covered hashing, symmetric and asymmetric encryption, we'll be in a good place to start discussing hybrid encryption. This is where you use the benefits of each of these approaches to create a secure encryption scheme. This involves use of a random number to create a single-use symmetric encryption key. This is what I

use to encrypt some data, but then that key is also encrypted using RSA to get the benefits of the public and private key system. Hashing is then used to perform integrity checks against the data so that a recipient of the data can tell if that data has been corrupted or tampered with in transit. This module will walk you through a code demo that results in a class library that you can use straight off the back of this course for performing hybrid encryption in your own applications. This course contains a set of code samples for each module. All of the code for this course, apart from a few small cases, runs cross-platform across Windows, macOS, and Linux. You can use Visual Studio for Windows or Visual Studio for Mac. The Community versions are fine. You can also use the fantastic JetBrains Rider IDE to compile, build, and run all of the code. If you prefer to use something a little lighter weight, then you can, of course, use VS Code, providing you have the relevant C# compilation and debugging plugins installed. All of the code for his course was written in C#, but the concepts and libraries used in the course are available in any .NET language like VB.NET. The cryptography techniques discussed in this course will reside in the system.security.cryptography namespace in the .NET Framework base class libraries. You can get all of the latest versions of the source code for this course on GitHub. On the screen, you can see a link to my GitHub page at github.com/stephenhaunts. On there you can find a repository called Building-Secure-Applications-with-Cryptography-in-.NET-Course-Source-Code, or you go to the Bitly link on the screen, and that will take you directly to the correct repository. So let's now take a look at what cryptography actually is.

What Is Cryptography?

Before we proceed to the main content of this course and look at practical cryptography in .NET, let's first take a look at some background information about cryptography itself. The most common use case for cryptography is that of making some information secret. This is done using a process called encryption. In this example, we have a message that reads Meet under the clock tower at 12 pm. Wear a red rose on your lapel. Through encryption we want to turn that message into an unreadable form so that it can be sent to a person who wants to read it. To do this, you encrypt the message with a secret key. Only you and the recipient know that key. Once the message has been encrypted, you can send that message to the recipient. The message they receive will be in a completely unreadable form. Before they can read the message, it will need to be decrypted. To decrypt the message, you need to run the message with the same algorithm using the secret key that is known to both parties. If they have the same key, then the contents of the message will be revealed. This is just one example of encryption. The type of encryption described here is called symmetric encryption because the same key is used to both encrypt and decrypt the message, provided both the sender and the receiver have that same key. Symmetric encryption is powerful, but it does come with its own set of challenges, in that

you have to be able to securely share the key with multiple parties, which is easier said than done. To help mitigate this problem, there's another type of encryption called asymmetric encryption, which we'll discuss in more detail later in this course.

Security Concepts

Cryptography covers many different types of security operation. There are four main areas that are covered by cryptography, and we'll be covering each of these areas in this course. The first, which is what we have just demonstrated, is that of confidentiality. Confidentiality is what you traditionally associate with cryptography. This is where you take a message or some other data and encrypt it to make the original data completely unreadable. There are many different cryptography algorithms that you can use, but this course will cover the main primitives that are use today, including RSA and the advanced encryption standard. The second type of security operation is that of integrity. In information security, data integrity means maintaining and assuring the accuracy and consistency of data over its entire lifecycle. This means that the data cannot be modified in an unauthorized or undetected manner. Integrity is violated when a message is actively modified in transit. Systems typically provide message integrity in addition to data confidentiality. We'll cover some different cryptography primitives that you can use to help enforce data integrity, including hashing techniques such as the secure hash algorithm. We'll also cover hash message authentication codes. The third type of security operation is called non-repudiation. Non-repudiation is the assurance that someone cannot deny something. Typically, non-repudiation refers to the ability to ensure that a party to a contract or a communication cannot deny the authenticity of their signature on a document or the sending of a message that originated with them. On the internet, a digital signature is used not only to ensure that a message or document has been electronically signed by the person that purported to sign that document, but also, since a digital signature can only be created by one person, to ensure that a person cannot later deny that they have furnished the signature. In this course we'll cover digital signatures using the RSA cryptographic primitive. The fourth and final security operation is that of authentication. Using a cryptographic system, we can establish the identity of a remote user or system. A typical example is the SSL certificates on a web server, providing proof to a user that they are connected to the correct server. The identity is not of the user, but the cryptographic key of the user. Having less secure key lowers the trust that we place on the identity. Towards the end of this course we'll be combining all of the primitives that we learn about using a technique called hybrid encryption. To demonstrate this, we'll build up an example of two people who are talking to each other over an encrypted channel, much like TLS. We'll deal with the problem of communicating with strong encryption with a good key exchange procedure and

the ability to detect any tampering of the message during the communication, and also be able to prove that the message came from the correct sender. In this module, we first talked about the audience for this course, who are .NET developers wishing to secure data within their applications using the built-in cryptography libraries in the system.security.cryptography namespace. We then looked at what this course will cover. This includes random numbers, hashing and authenticated hashing, secure password storage, symmetric encryption, asymmetric encryption, hybrid encryption, and digital signatures. The demo code for this project is written in C# and works within the .NET Framework, .NET Core, .NET 5, and the future. You can use Visual Studio on Windows or the Mac, as well as JetBrains Rider and VS Code for Windows, Apple macOS, and Linux. We then took a very quick look at what cryptography is and gave a simple example of symmetric encryption using a shared encryption key. This then led onto a discussion about some of the key security concepts that cryptography tries to address, which are confidentiality, integrity, non-repudiation, and authentication. Now you have completed this introductory module, let's start our practical exploration of cryptography in .NET by taking a look at cryptographic random numbers.

Cryptographic Random Numbers

Overview

Welcome back to my course, Building Secure Applications with Cryptography in .NET. In this module, we'll cover how to generate random numbers that are suitable for use for cryptographic keys. What you will learn in this module will be used throughout the remainder of this course as it forms a fundamental underpinning to everything that we do. First of all, we'll take a look at why random numbers are so important in the field of cryptography. Then we'll take a brief look at the System.Random random number generator in .NET and discuss why it is good for some circumstances, but it is not suitable for use with cryptography and security. We'll then take a look at the RNGCryptoServiceProvider random number generator in .NET, which is recommended for use. Once we have covered why RNGCryptoServiceProvider is a better solution, we'll take a look at a code demo running in Visual Studio. So, let's get started and look at why random numbers are so important.

Why Are Random Numbers So Important?

Most encryption algorithms require a source of random data to generate their encryption keys. Most computers do not have a hardware-based random number generator available, so programmers have had to resort to software-based

techniques to generate random numbers as best they can. Because these random numbers are generated in software, they are very rarely truly random. They're typically pseudorandom, in that they appear random, but they are not. To generate this random data, you need a source of entropy or random inputs. Some software random number generators will ask you to move the mouse or type on the keyboard as a source of that entropy. Others will take events such as hard drive activity or network activity for that source of entropy. This typically works quite well for end-user workstations, but can become a problem in some cases, such as a diskless network appliance with a network card, some flash RAM, and the CPU. A network appliance that performs encryption has no good sources of entropy, apart from network events, which an attacker could manipulate to their advantage. You can also use a hardware device to generate a secure random number. These devices are called hardware security modules. But for the purposes of this course, we want to use a random number generator that's built into the .NET Framework base class libraries. Before we look at an implementation of a secure random number generator, let's take a look at the more common System.Random class that exists within .NET.

System.Random and It's Problems

In .NET, when you need to generate a pseudorandom number, you might use the System.Random class to generate that number. The number generator works by providing a seed value when the object is constructed. For most application scenarios, this is fine, and we'll get the appearance of randomness when you apply a different seed every time. When you are dealing with security though. System.Random is not sufficient. As a result, is deterministic and predictable. The current implementation of the System.Random class is based on Donald E. Knuth's subtractive random number generator algorithm taken from his book, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, published by Addison-Wesley. In his book, he states a subtractive number generator calculates a sequence of random numbers where each number is congruent to the subtraction of the two previous numbers from the sequence. So we know that System.Random is not useful as a tool for creating cryptographically secure random numbers due to it being deterministic and predictable. So let's look at a good solution for creating good, secure random numbers, and that's using the class RNGCryptoServiceProvider.

Secure Random Numbers with RNGCryptoServiceProvider

Random numbers are very important in cryptography. You need them for generating encryption keys for symmetric algorithms such AES and for writing randomness to hashing functions and key derivation functions, which we'll cover

in later modules of this course. As we've just discussed, System.Random is not very good for generating non-deterministic random numbers. The .NET base class library has a mechanism built into it to help us generate cryptographically secure random numbers. We will use the RNGCryptoServiceProvider class to generate the random numbers that we need for our cryptographic applications. This is a much more secure way of generating these numbers. The tradeoff to these much more secure random numbers is that RNGCryptoServiceProvider is much slower to execute compared to System.Random. This is a small tradeoff though when you want to do random numbers being generated to use these encryption keys. So let's now take a look at a demo of how to use RNGCryptoServiceProvider.

RNGCryptoServiceProvider Demo

In the demo, we'll introduce a class that contains a static method to generate a random number two a specific length in the byte array. The demo will use this class to generate 10 different random numbers. The code we introduce in his first demo will be something that we'll need to use throughout the remainder of this course. So, let's get started. On the screen now you can see in Visual Studio that we have the Solution file loaded up that was available in GitHub. The address for this repository in GitHub was shown in the introductory modules of this course. Our Solution file we split down into lots of different solution folders, which correspond to how we're going to teach the content in this course. Each demo within this Solution file has been built as a completely self-contained console application so that everything can run independently from each other. So what we're going to do first is we're going to look in the Random Numbers solution folder. In here, we have a project called RandomNumber. So first of all, let's open Random.cs. So in this file we have a class called Random that we've built, which is a static class. Inside this class, we have a static function called GenerateRandomNumber. As a parameter, we pass in a length, which is going to be a length in bytes that we want our random number to be. And then from this method, we're going to return a byte array, which corresponds to the length that we pass in. So first of all, we create a new instance of the class RNGCryptoServiceProvider, and we then create a byte array corresponding to the length that we pass in as our first parameter. So for example, if we wanted to create a 32-byte array full of random data, then we'd pass 32 into our parameter. Next, we call the GetBytes method on our randomNumberGenerator instance, and we pass in the byte array that we've just created. This will populate that byte array full of random data, which we then return to our caller. So if we look in Program.cs, we can see here that we're going to very simply create a for loop, which we're going to go around 10 times and you're going to generate 10 random numbers to which we're going to display to the console. To display these random numbers to the console, we first need to turn that byte array into something that's more readable on the screen. So what we do is we call the

ToBase64String method on the Convert class, and what this will do is it will turn the byte array into a Base64 encoded string, which makes it easier to either display on the screen or store in a database, for example. So let's run this demo and see what happens. So what we're going to do is we're going to start off in our loop on the first iteration, and then what I'm going to do is I'm going to step into our GenerateRandomNumber method. So first of all, we create the instance of RNGCryptoServiceProvider, we then create our byte array, in this case it's 32, so I want this array to be 32 bytes long, which as we can see here it is, but at this point the array is empty, it's just full of 0s. So we then call GetBytes on our randomNumberGenerator, and at this point if we look inside our byte array, we can see that it's completely full of random data. So we'll now pass this back to the caller, and we can then go ahead and run that 10 times, but I'll just run this to the end. As you can see in the console window, we have displayed our 10 random numbers onto the screen, but they are represented as Base64 encoded strings. So as we discussed, generating random numbers is absolutely fundamental to everything we need to do in this course to create secure encryption keys. But from what you can just see with code, it's actually very straightforward to do, which is very good.

Summary

In this module, we took a look at random number generation. We first looked at the System.Random class and showed that because of its deterministic and predictable nature, it is not suitable for generating random numbers for cryptographic purposes. We then looked at the RNGCryptoServiceProvider class, which is designed for generating truly random numbers that can be used for generating encryption keys. We introduced the RNGCryptoServiceProvider class in a code demo. We'll be using this class many times throughout this course. In the next module, we'll look at some of the different hashing operations within the .NET Based Class Libraries.

Hashing Algorithms

Overview

Welcome back to my course, Building Secure Applications with Cryptography in .NET. In this module, we're going to take a look at hashing and hash message authentication codes to satisfy our integrity and authentication pillars in cryptography. We'll first start by taking a look at what hashing actually is and what it means to hash data. We'll then take a look at the MD5 hashing algorithm and why it is not recommended for use these days. Then we'll take a look at the Secure Hash family of hashes, including SHA-1, SHA-2, and SHA-3. At this

point, we'll then look at a code demo of how these algorithms work in use. Then we'll take a look at extending the hashing concept by bringing in a level of authentication with hashed message authentication codes, or HMACs for short. This will be followed up with another code demo exploring the use of HMACs in .NET. So now let's take a look at what hashing is and why we need it.

What Is Hashing?

Hashing is a fundamental primitive used within cryptography. In this section, let's take a look at what hashing actually is. A cryptographic hash function is an algorithm that takes an arbitrary block of data, passes it through a hashing function, and returns a fixed size block of data, the cryptographic hash value, such that any accidents or intentional change to the data will change the hash value. The data to be encoded is often called the message, and the hash value is often called the message digest, or simply the digest. The ideal cryptographic hash function has four main properties. First, it is easy to compute the hash value for any given message, and then it is infeasible to generate a message that has a given hash. What this means is that you should not be able to pick a particular hash value and create the original message that generates that hash. Next we have it's infeasible to modify a message without changing the hash. What this means is if you generate a hash for a particular message, if you change just one bit of the original message and recalculate the hash, the resulting hash should be totally different to the original hash message. And finally, it should be infeasible to find two different messages with the same hash value. This is also referred to as a hash collision. A good hash function should guard against being able to generate the same hash code or digest for two different messages. Another way of thinking of a hash function is that of creating a unique fingerprint for a piece of data. For any data that you pass through a hash function, its digital fingerprint should be unique. If the data changes, even just by one bit, then the fingerprints will be completely different. Generating a hash of data in .NET is very easy to do as the remainder of this module will demonstrate. As with the module on random numbers, what you'll learn in this module will be required later when we talk about hybrid cryptography. In the .NET Framework, .NET Core, .NET 5, and future versions, there are various different hashing algorithms that you can use, such as MD5 and the Secure Hash family, such as SHA-1, SHA-256, and SHA-512. As you'll see when we look at the code demos, each of the hashing algorithms described here share a Base Class Library called HashAlgorithm. This means that the usage of each algorithm is identical, which makes them very easy to use. A hash function is a one-way function. That means once you have hashed some data, you cannot reverse it back to the original data. On the flipside to this, encryption is designed to be a two-way operation. Once you have encrypted some data using a key, you can reverse the operation and decrypt the data by using that same key. When you hash some data, the hash will be the same every time you perform the operation unless the original data changes in some way. Even if the data only

changes by one bit, the resulting hash code should be completely different. This makes hashing the perfect mechanism for checking the integrity of data. If you look at the screenshots on the screen at the moment, there are two strings that have been defined. Message 1 reads Original Message to hash. The second message also reads Original Message to hash, and it differs by only one character. Here you can see that we have changed character 3, which is a lowercase i, to a 1. If you look at the hashes of both messages on the screenshots, they're both completely different, even though the message only differs by one character. Hashing is useful when you want to send data across a network to another recipient, and that recipient wants to check whether the data that was sent is intact and has not been tampered with or corrupted. Here we have Bob on the left of the screen. He has a message that he wants to send Alice over the internet. The message reads, Meet me at the coffee shop to exchange contracts. Before sending the data, Bob calculates a hash of the data to get his unique fingerprints. He sends that data and the hash over to Alice. Alice then recalculates the hash at the message that Bob has just sent, then compares that hash to the hash it was sent with the original message. If the hash codes are the same, then the data has been successfully received without any data loss or corruption. If the hash codes do not match, then the data received is not the same as the original data that was sent by Bob. This means that Alice cannot trust that message sent by Bob as it doesn't generate the same hash.

MD5 and Secure Hash

The two most common hashing methods used are MD5 and the Secure Hash family of hashes, such as SHA-1, SHA-256, and SHA-512. Let's explore them in more detail. MD5 was designed by Ron Rivest at the RSA Security Company in 1991 to replace MD4, an earlier hash function. The MD5 message-digest algorithm is a widely used cryptographic hash function that produces a 128-bit, or 16-byte, hash value, which is typically expressed in text form as referred to by a hexidecimal number, or a Base64 encoded string. MD5 has been used in a wide number of cryptographic applications and is also commonly used to verify file integrity. In 1996, a flaw was found in the design of MD5 where it was possible to detect hash collisions where parts of the hash could be the same across multiple messages. While it was not deemed a fatal weakness at the time, cryptographers began to recommend the use of other algorithms, such as the Secure Hash family. In 2004, it was shown that MD5 contained further hash collision resistance problems, which means that it is possible to generate an MD5 hash of two sets of data, which could result in the same hash, although this was quite rare. You may still need to use MD5 in your applications if you're checking the integrity of data coming from a legacy system that makes use of MD5. So it is still relevant to discuss MD5 in regard to legacy application support. The Secure Hash family is a family of cryptographic hash functions published by The National Institute of Standards and Technology, also known as NIST. The Secure Hash family comes

from different variants, including SHA-1, which is a 160-bit hash function, which resembles the earlier MD5 algorithm. This was designed by the National Security Agency to be part of their digital signature algorithm. Cryptographic weaknesses were discovered in SHA-1, and the standard was no longer approved for most cryptographic uses after 2010. Then we have SHA-2, which is a family of two similar hash functions with different block sizes known as SHA-256 and SHA-512. They differ in their word sizes. SHA-256 uses 32-bit words where SHA-512 uses 64-bit words. These versions of the Secure Hash family were also designed by the National Security Agency. And then we have SHA-3, which is a hash function based on the Keccak cypher, which was chosen in 2012 after a public competition among non-NSA designers. It supports the same hash lengths as SHA-2, and its internal structure differs significantly from the rest of the Secure Hash family. SHA-3 is not currently supported in the .NET family directly, although third-party implementations are available. Implementing SHA-1 and SHA-2 in your applications is very straightforward to do, as you'll see in the following demos. We'll not be exploring SHA-3 in this course because it is not currently a part of the .NET Base Class Libraries.

Hashing Demo

In the following code demo we'll take a look at MD5, SHA-1, and SHA-2 in action. For SHA-2, I'll show you how to use both the 256 bit and the 512 bit variants. So let's take a look. If we go back to our Solution Explorer and open up folder number 2, which is called Hashing, in here we have a number of projects. The project we want to look at is called Hashing. So first of all, if we look in the Hashing.cs we have a static class called HashData. And inside of here I have a small, static method for each of the different hashing types. So we have one for sha1, we have anther one for sha256, sha512 and then at the bottom here md5. So let's just pick sha256 for the moment. So in this method, we take in a byte array, which is our data that we want to be hashed. So if we were to pass in a string, we'd have to convert that string into a byte array first. The same as if you load up a file, when you load up the file it will be a byte array. Now this method also returns a byte array, which will be our hashed message or our hashed data. So in this method, for example, we have on object with sha256, and we call the static Create method on that. And that creates an instance of our hashing algorithm. And to calculate a hash for some data that we pass into that method, we just call the ComputeHash method on that sha256 object. Once we have done that, we return the result to the caller. So to generate the sha256 hash it's very straightforward. It's just a few lines of code. Now if we compare that to, say, md5 at the bottom here, you can see that the interface is exactly the same. The only thing that's different is the class name is called md5 and not sha256. And the same goes for sha512 and sha1. So they're all virtually identical to use except for their actual class name. So if we go to our Program.cs example code, what we're going to do in this demo is we have two strings, we have originalMessage

and originalMessage2. So the first one says Original Message to hash, and the second one says exactly the same thing, but like before we've changed the third character from a lowercase i to a 1. And what we're going to do is in the demo we're going to calculate an MD5 for both of those messages, a SHA-1 for both of those messages, and then the same for SHA-256 and SHA-512. And what you'll see is each time we generate that hash message twice for each message, you'll see that the hash messages or the hash codes are actually different. So let's run this and step through it. So first of all, we start off, we just output the original messages to the screen. Then what I'm going to do here is I'm just going to step into computing the MD5 hash. Now before we can do that, because our message is a string, we need to convert it to a byte array first. So here I'm calling the Encoding.UTF8.GetBytes method, which will convert that string into a byte array. So the point when we come into our ComputeHashMd5 method, we can see that what we're passing in is now a byte array. So that's our string in byte array form. So first of all, we call the Create method on the md5 object in the base class library. Then we call ComputeHash. And then we return the result, which will be a byte array. And we also do the same for our second message. So now I want to do this for SHA-1. And I'm not going to step into all of these, but just to, again, illustrate the point, if we step into our ComputeHashSha1 method, it's very similar to what you've just seen with the MD5 method where we call the Create method on our objects, in this case it's called sha1, and we then call ComputeHash on the byte array that we've passed in and return the result. So if we now go and do this for SHA-256 and SHA-512, we're now at the point where we can start outputting our hashes to the screen. And as you can see, our hashes here they're a byte array, but we want to display it on the screen, so we need to convert it into something that's more human readable. So, like with the random numbers, we're going to use the Convert.ToBase64String to convert that byte array into a human readable string. We'll do that for our SHA-1, our SHA-256, and our SHA-512. And we display it on the screen. So if we go to our console window, you can see the results on the screen here. So as you can see here, we have our Original Message 1 and our Original Message 2 which differ by only one character. And then we have the MD5, SHA 1, SHA 256, and SHA 512 hashes for both of those different messages. Now even though the message only differs by one character, you can see that each of the hash code pairs is completely different from each other. And also the thing that you can see that's quite noticeable is that with each different hash type the size of the hash that we're generating is larger. So, SHA 1 is, you know, a little bit larger than MD5. But if you look at the difference between SHA 256 and SHA 512, you can see that the SHA 512 message is about double the size of SHA 256. So that's our look at straightforward hashing. Let's now take a look at hash message authentication codes.

Hashed Message Authentication Codes

If you combine a one-way hash function with a secret cryptographic key, you get what is called a hash message authentication code, or HMAC for short. Like with a hash code, a HMAC is used to verify the integrity of a message. A HMAC also allows you to verify the authenticity of a message because only a person who knows the key can calculate the same hash of that message. A HMAC can be used with different hashing functions like MD5 or the secure hash family of algorithms. The cryptographic strength of a HMAC depends on the size of the key that is used. And the most common attack against a HMAC is a brute force attack to try and uncover that key. HMACs are substantially less affected by collisions than their underlying hash algorithms, as the addition of the secret key adds entropy to the hashing operation. HMACs are use when you need to check for both integrity and authenticity of a message. For example, consider a scenario in which you are sent a piece of data along with its hash. You can verify the integrity of the message by recomputing the hash of that message and comparing it with a hash that you've also received. However, you don't know for sure if the message and the hash were sent by someone you know or trust. If you use the HMAC, you would recompute the HMAC using a secret key that only you and the sender know, and compare it with the HMAC that you just received. This is the purpose of authentication, which is one of the security pillars that we looked at in the introduction of this course.

Hashed Message Authentication Codes Demo

In the following demo, I'll show you how to use HMACs that are based on MD5, SHA1, SHA256, and SHA512. This demo will use the HMACMD5, HMACSHA1, HMACSHA256, and HMACSHA512 classes in the .NET Framework. If we're going go back to folder 2 in our solution, which is called Hashing, open that up, and then we have a project in here called HMAC. If we open up HMAC.cs, we have a class here called Hmac. Now the first thing we do is we have a static public method here called GenerateKey. This is exactly the same as what we saw in the previous module. So what our GenerateKey method would do in this instance is it will generate a random number, which is 32 bytes in length. Instead of passing in the size into GenerateKey, we instead have a private const, which is set to 32 bytes because this is the size of the key that we want. If we scroll down, we then have different methods for each of our different HMAC types, so if we first look at the first one here, which is ComputeHmacsha256. As parameters, we pass in two pieces of data. The first is a byte array of the data that we want to generate the HMAC for. So this could be a string or it could be a file or anything. And then we have a byte array, which is the key that we've just generated. When it comes to calculating the HMAC, we instantiate a class called HMACSHA256, and we pass the key in its constructor. Once that has been constructed, we have an object called hmac, which we can then call ComputeHash on where we pass in the data that we want to hash, and this will return a byte array of our computed hash, which we then return to the caller. Look

at the next method down, which is exactly the same, but for a SHA1 variance. It looks almost identical apart from the class name, which in this case, is HMACSHA1. And then we have exactly the same for HMACSHA512 and HMACMD5. So as you can see, once you know how to do one, you know how to do them all because the interfaces are the same. If we go back to our program.cs, this looks very similar to what we looked at in the hashing demo. So we have two strings, originalMessage and originalMessage2 where the second message differs by only one character. In this case, we have changed the uppercase M in Message to an x. Then the first thing we do is we generate our key. So we call HMAC.GenerateKey, which gives us a variable here called key, which contains our 32 bytes random number. Then what we're going to do, similar to the hashing examples, I'm going to calculate two hashes for each variant, one for originalMessage and one for originalMessage2, and we'll then display those messages on the screen. So if we look at the first one here, we are computing a HMACMD5. So we, first of all, convert our message into a byte array using the Encoding.UTF8.GetBytes method. And we also pass in our key, which is the key we want to use to calculate the HMAC. And we do this for MD5, SHA1, SHA256, and SHA512 variants of the HMAC. And we then output all of these onto the screen. So let's run this and take a look at what happens. So first of all, we generate our key. And as I said before, this is exactly the same as what we looked at in the previous module. So we create an instance of RNGCryptoServiceProvider, instantiate a byte array to the desired size, which, in this case, is 32 bytes. We then call GetBytes on the randomNumber generator, and we can then return our randomNumber to the caller, which is in this variable here called key. So now we're going to create a HMAC for an MD5. So, first of all, we take our original message, which contains the message originalMessage to hash. We then create that into a byte array using our Encoding.UTF8.GetBytes method, and we also pass in our key into the HMACMD5 or ComputeHmacmd5 method. So if we step into that, we can see we have our byte array of our data, and we have our 32-byte key. So we instantiate the HMACMD5 class whilst passing the key in as the constructor object. And we then call ComputeHash and return the result. So we do this for our second variant of the message. Let me do this for SHA1, SHA256, and SHA512. We then go and output all of these onto the screen. So again, we're using the Convert.ToBase64String. So if we go and write these to the screen So here we can see our HMACs have been printed to the screen. So conceptually, very similar to what we looked at earlier in the module with hashing, but the addition here is that we provide a key to the hashing method.

Summary

The ability to generate a hash code for some data is an important thing to be able to do as it helps to satisfy the integrity requirement that we discussed earlier in the course. By generating a hash code for some data that we sent across the

network, the recipient can also generate the hash code for the data they receive and compare it to the original value. If the values match, the data is intact and has not been tampered with. If the value does not match, then the data can be disregarded. Hash message authentication codes offer a similar service, but they differ in the keys used when creating the hash code. If a recipient was to generate the same hash code for the data, they also need to be in possession of that key. In the next module, we'll take a look at the use of hashing for protecting passwords

Secure Password Storage

Overview

Welcome back to my course, Building Secure Applications with Cryptography in .NET. In this module, we'll talk about storage of passwords in your systems. Passwords are still the most common way of being able to authenticate a user, but it's very easy to put yourself in a situation where your system is not secure and susceptible to attacks. We'll start off by discussing techniques that you shouldn't use and gradually iterate to a better solution. We'll look at storing passwords in the clear, encrypting passwords, using hashes to store passwords, using salted hashes to store passwords, and finally, using password-based key derivation functions.

How Not to Store Passwords

When you're developing a system that needs to authenticate a user and set up passwords, you need to store those passwords somewhere that the user can come back later to log in. Generally, this might be a SQL or a NoSQL database. There are various techniques you can use to encode the password, but the biggest mistake that you can make is storing passwords as clear text in your database. To do so counteracts the very benefits of having passwords in the first place, which is to authenticate and grant access to a system using the password known only by the user. Once a system needs to be able to verify a password, that system should never need to know what the actual password is. A big risk here is that if your database is ever compromised and the password tables are stolen, the attacker doesn't have to do much hard work to gain access to your customers' accounts. They just log in with the username and the passwords that were stolen from your database. This could have catastrophic consequences for your business. We'll be covering encryption and decryption in more detail later in his course, but I want to talk about encrypting passwords now. As we discussed in previous modules, encryption is a two-way process .You can create using a key and then decrypt with same key. This is called symmetric encryption. This

mechanism could be used to store passwords, and it's certainly much better than storing passwords in the clear. But this approach comes with its own set of problems and challenges. If the password tables are compromised as before with our example of passwords being stored in the clear, the attacker will have access to a full table full of encrypted passwords. They might be thinking this is good because they can't see the actual passwords. Well, yes, it is. But you also have to manage the encryption keys used to store those passwords. Where do you store those keys? What happens if the key has been stolen? Do you then have to decrypt and re-encrypt all of your passwords with a new key? While the passwords themselves are encrypted so that it cannot be read, the overhead of managing the keys is actually quite a big headache. Instead of encrypting the passwords, hashing passwords has also been a common technique, where you take a password, hash it with something like SHA-256, and then store that hash in the database. As we discussed earlier in the course, hashing is a one-way function, so you shouldn't be able to reverse engineer the password back from the original hash. Whilst hashing the password with an algorithm like SHA-1 or SHA-256 seems like a much better solution, it does have some shortcomings. Hashing a password is susceptible to two different attacks, that is, a brute force attack or a dictionary or rainbow table attack. A brute force dictionary attack is where the attacker will try a different combinations of passwords until they get one that matches the password hash. This is easier if the attacker has already compromised your hash password tables from your database. To try and make passwords stronger, people tend to try adding in different characters, like turning vowels into numbers in the hope that this will make their password stronger. But it really doesn't make that much difference to anyone doing a brute force attack. It may sound like a lot of work to try and break a password this way, but with the advent of modern processors and graphics processing units, it's possible for an attacker to try billions of passwords every second. It's only a matter of time until they get the correct password. Unfortunately, people still tend to pick weak passwords based on names and other simple words, so a brute force approach like this is remarkably effective. An alternative attack from a brute force attack is that of a rainbow table. A rainbow table contains a large dictionary of precomputed hashes for different passwords. These tables are used to determine the original plain text of a password from already precomputed password hashes. Rainbow tables can be many gigabytes in size, and they greatly speed up attacks to recover the original value of a password. A better approach is to increase the entropy of the password being attacked by making the password harder to recover. Entropy, in this case, is a measure of a password strength and how easy it is to guess. If you extend the password of additional random letters and numbers, then this is much harder to get an attack by either using a brute force attack, a dictionary attack, or a rainbow table attack. This type of storage mechanism can be done by adding a salt onto the password before hashing. The salt can be generated as a random number and then appended to the password before hashing. By adding a salt, you're making the password a lot more complex, and this makes a brute force or

rainbow table attack much harder. To check if a password is correct, we need the salt, so it's usually stored in the database along with the hash. An attacker won't know in advance what the salt would be, so they can't precompute a lookup table. Each time you hash a new password, you should use a new salt value. This means that if two users have the same password, because the salts are different each time, the hashed password would look completely different, even though the passwords are the same. Salting a password like this is a very common technique used today by companies all over the world, but you can go one step further. Let's take a look.

Password Based Key Derivation Functions

The problem with hashing and storing a password, even with salt, is as processes get faster over the years, we run the risk of what we currently think of secure passwords being compromised because the processes can perform a brute force attack and a rainbow table attack much faster. What we need is a solution that allows us to still hash our passwords with a salt, but it helps us guard against advancements in Moore's law. Moore's law, which is named after Gordon E. Moore, cofounder of Intel, states that over time the number of transistors in a circuit will double approximately every two years. This means that as processors and graphics processors become faster and more powerful, what may be a difficult password to brute force attack today may be very easy in a few years as more combinations can be tried in a shorter space of time. The solution here is use what is called a password based key derivation function, or a PBKDF2, as it is also known. A PBKDF2 is part of the RSA Public Key Cryptographic Standards series, otherwise known as PKCS #5 v2.0. The part of a base key derivation function is also part of the Internet Engineering Taskforce's RFC2898 specification. All of this terminology makes this process sound a little scary and hard to use, but in reality, apart from the intimidating name, this method of hashing passwords is actually very easy to use in .NET. A password based key derivation function takes a password, a salt to an additional entropy to the password, and a number of iterations value. The number of iterations value repeats a hashing operation over the password multiple times to produce a derived key for the password that can then be stored in the database. By repeating the hashing process over the password multiple times, you are algorithmically slowing down the hashing process, which makes brute forcing against attacks of the password much, much harder to do. This means that fewer password brute force attempts can be done at once. As processes get faster over time, you can increase the number of iterations used to create the new password hash, which means password based key derivation functions can scale with Moore's law. A good default to start with a number of iterations is around 100,000. This means when you call the password based key derivation function, the password will be hashed 100,000 times before it's returned to you. You need to pick a number of iterations that gives you suitable security

versus performance for your system. For example, if you're using a password based key derivation function as part of the user authentication with your system, then it's the additional delay encountered for the chosen number of iteration cycles acceptable. For example, if you log into an ecommerce website, you may notice that sometimes, once you've hit return after entering your password, there might be a delay for a couple of seconds. The chances are that there is a password based key derivation function, or something similar, happening behind the scenes to rehash your password. You need to pick a number of iterations, which introduces a delay as acceptable to your needs. Ideally, this number of iterations factor needs to be changed and preferably doubled at least every two years. You can manage this as part of your standard password updating process. If you force your staff or customers to update their passwords every x months, then you can design your system from a specific point in time when a new password is hashed and stored to use the larger number of iterations. By also adding the salts to your derivation function, you further reduce the ability of a rainbow table to be used to recover the original password. The salt should be unique for each password. As we discussed previously, the salt is not a secret value, so it can be stored in the database along with your hash password. A good minimum length for your salt is at least 64-bit, or 8-bytes. But in our demo in a moment we're going to use a 256-bit, 32-byte salt. The .NET class that we're going to use to perform the key derivation function is called Rfc2898DeriveBytes. When constructing this object, you pass in the data to be hashed as a byte array. The salt is a byte array and the number of iterations you want the algorithm to perform. The salt in our code demo is generated using the RNGCryptoServiceProvider class as previously discussed in the module on secure random number generation. Remember, the salt doesn't have to be kept secret and can be stored alongside the hash password, and a salt's purposes to give added entropy to the password being hashed. So let's go and look at a demo.

Password Based Key Derivation Functions Demo

In the following code demo we'll use the RFC2898DeriveBytes class in .NET to hash a password to a different number of iterations. We'll also use a Stopwatch object to time how long these different variations take. So let's get started. In the Solution file for the example code, I want you to, again, go into Solution folder 2 called Hashing. And in here I have a project called PBKDF2. So if we open this up, we have a two files in here. So first of all, let's look at PBKDF2.cs. Now in here, I have a public static method called GenerateSalt. Now what this is doing is, it's generating a 32-byte random number, so that's a 256-bit random number in total. Now this is what we're going to use to generate our salt. So this code is pretty much identical to what we've seen earlier in the course, and it returns a byte array which contains our 32 bytes of random data. Now the actual code for performing the password-based key derivation function hash is incredibly

simple. It's just a few lines of code. So let's look for it. So we have a method, a public static method, called HashPassword. Now this method takes a byte array, which is the password we want to be hashed. So again, if your password is a string, we need to convert it to a byte array first. It takes a byte array containing our salt, and it also contains a numberOfRounds, which is our number of iterations, so that's how many times we want the password to be hashed. Now to use this, it's very simple. So we have a class called Rfc2898DeriveBytes. Now this is a bit of a strange name for a class, so if you're quite familiar with RFC specifications numbers, then this makes complete sense. It's RFC2898, which is a specification for a password-based key derivation function. Now if you've never come across that specification before, which I hadn't when I started learning this stuff, this class is very easy to just easily gloss over in the MSDN documentation because it's a bit of a strange name, so you don't really think much of it. But actually, this is incredibly powerful, and as a tool for hashing passwords, it is incredibly fundamental and easy to use, which is fantastic. So what we do is, we construct an instance of Rfc2898DeriveBytes, and we pass in our data to be hashed, we pass in our salt, and pass in numberOfRounds. Then to generate the hash we call GetBytes on our rfc2898 object. Now you'll notice that we pass in 20. So we're saying we want it to generate and retrieve 20 bytes. The reason for this is because Rfc2898DeriveBytes internally uses SHA1 as its hashing algorithm. Now SHA1 only gives you a 20-byte hash, so there's no point requesting any more than that. It's just 20 bytes that we want. But at the point when we get that 20-byte array back, if you passed in, say, 100,000 into your numberOfRounds, that SHA1 operation's going to be called 100,000 times before you get those 20 bytes back. So let's look at Program.cs, which is our example. So what I have here is, I'm going to be hashing a password, so I've got a password here called VeryComplexPassword. Also a very terrible password, but is fine for illustration purposes. Now I have a little helper method here that we'll look at in a second. I'm going to hash its password multiple times, but I'm going to start off with 100 iterations, then 1000, then 10,000, 50,000, 100,000, 200,000, then half a million. What we're going to do, as you can see here is, we're going to set a Stopwatch, and that Stopwatch is going to time how long it takes each of those password-based key derivation functions to execute. So what we should see is, as the number of iteration counts goes up, it's slower to retrieve the password. So let's just step into this and take a look at what's going on. So we're going to step into our HashPassword helper method here, and we start the Stopwatch. Now, the Stopwatch aspect at the moment is kind of pointless because I'm trapped in the debugger so the times are going to be all over the place, so we will rerun it. And what I've done is I've stepped over our HashPassword method in the PBKDF2 object, and you can see we've retrieved a 20-byte password back. We step into that individually, so we here we're creating our salt, and then I've constructed the Rfc2898DeriveBytes object, I've passed in our password to be hashed, the salt, and the numberOfRounds. In this case, it's 50,000. And then I call GetBytes(20), and that's going to have done 50,000 SHA1 hashes before it gives us the password

back. And what I'm going to do is, I'm going to stop running this because we're dependent on the stopwatch. Being trapped into the debugger has completely thrown all the timings off. So let's remove the breakpoint and then just let it run. Okay, so we've got some interesting timings here. So the very first iteration was 100. Now it's given us 103ms, but it's probably because .NET was doing some precompiling behind the scenes, so let's ignore that one and start with the next one, which is 1000 iterations. So that operation took 2ms to calculate, Then the next one, which is 10,000 iterations, took 6ms; 50,000 iterations took 53ms; 100,000 seemed to be a bit faster and it took 49ms, for some reason; 200,000 iterations took 125ms; half a million took 362ms. So generally, apart from a few little edge cases there, you can see the actual process of increasing the integration counter slowed down the hashing process, which is exactly what you want. So now I'm running this currently on a fairly powerful 6 core machine, so it didn't take too long to run those iterations. But if you imagine setting your iteration count to a million or 2 million or 5 million, then you can see that this is going to quite drastically start slowing down that iteration process. So in this case, 362ms in computing terms is an eternity. Now, just before we end this demo, I just want to make you aware of some other code. So If we look in the HashPassword project up here, as I've said, using a PBKDF2 is kind of the preferred technique to use if you can. But what I've done in here, just for illustration purposes, I have done an example of doing a standard salted hash. So if we go into hash.cs, we can see we've got a GenerateSalt method here, which, again, provides us a 32-byte random number. And then I have a method here called HashPasswordWithSalt. Now what's happening here is, I'm taking a password to be hashed as a byte array. I'm taking the salt. I create a SHA256 object where I call the static Create method on the SHA256 object, just like we looked at in the last module. And then I call ComputeHash. But you can see what I'm doing here is, I'm calling a little helper method here called Combine, and I'm passing in two byte arrays. The first one is our password to be hashed, and the second one is the salt. And what this is doing is, it's creating one byte array with both of those two arrays combined. So what that effectively means, if you imagine they weren't byte arrays, and that they were strings, I've effectively just combined the two strings together. But in this instance, I've got two byte arrays, and I just copied them together. So if we look at the example code for this, and it's very simple. We have a VeryComplexPassword here. I create a salt or generate a salt, and then what I do is I call my HashPasswordWithSalt. I convert the password into a byte array. I also pass in the salt, and that returns the salted password for us. So I'm not going to step through that in any detail, but I will just quickly run it just so you can see the output. So we've got the password, which is VeryComplexPassword. We have the salt, which I've converted to Base64 string and put on the screen. And what we do is, we got that complex password and the salt, combine them together, and then hash those as one unit, which gave us our Hashed Password that you can see on the screen there. So whilst a password-based key derivation function is a preferred way of doing this in .NET, I've just thought it'd be interesting to include the salted hash version just for your general

information.

Summary

In this module, we've talked exclusively about securely storing passwords. We started off by discussing how it is a very bad idea to store passwords in the clear in your database. If your password tables were ever stolen by an attacker, then they'll very easily be able to gain access into your systems. We then looked at encrypting passwords as an alternative. Whilst this is a better solution, it does come with its own set of challenges, and you have to deal with the management and the storage of the cryptographic keys, be they symmetric or asymmetric. We also discussed how encryptions are a two-way operation where an encrypted password could be decrypted, but you should never actually need to be asked to decrypt a password. This led on to a discussion of using hash functions like SHA256 to encode your passwords. Unlike encryption, which is a two-way operation, hashing is a one-way operation only, meaning you shouldn't be able to divide the original password from the hash. Straight hashing of passwords does open you up to two types of attack, though. The first type of attack is a brute force dictionary attack, where an attacker we use a large dictionary of words and various combinations and substitutions to try and recreate a hash password. Once I get a matching hash, this means a part of it's been filled. The second type of attack is a rainbow table attack where an attacker uses a large database of pre-computed hashes to work out the password that created the hash. We then discussed a better version of password hashing that adds a random salt value to the password before its hash to add entropy onto that password. This salt value should be unique for each password hashed. This means that if two users have the same password, because I used a different salt, the hash password would look totally different. The hash value doesn't have to be kept secret, so it's okay to store along with the hash password in the database. We then took the salted hash concept one step further by introducing a password-based key derivation function. This is very similar to the salted hash, except that it incorporates a third element, which is a number of iterations value. This number of iterations value is designed to make the hashing process algorithmically slower. This is required as current modern processes and GPUs can process tens of billions of operations per second. So as hardware scales with Moore's Law, it means passwords will be easier to brute force over time. By making the hashing operation algorithmically slower, this limits the amount of passwords that can be checked in a fixed period of time. In the next module, we'll start taking a look at symmetric encryption and the different algorithms that are available to use in the .NET Framework base class libraries.

Symmetric Encryption

Overview

Welcome back to my course, Building Secure Applications with Cryptography in .NET. In this module, we're going to talk about symmetric encryption. We're first going to talk about what symmetric encryption is. Then we'll briefly talk about the DES and Triple DES algorithms. These are not recommended for use in new systems, but it is useful to know about them when you need to work on an older legacy system. Next, we'll talk about the Advanced Encryption Standard, which is a preferred encryption algorithm to use these days. Primarily, we'll be working with a mode called cipher block chaining. Next, we'll talk about two new AES encryption modes called GCM and CCM, and these were introduced into .NET with the .NET Core 3.0 release. If you're building a new system, then GCM and CCM are the modes that you might want to look at. Finally, we'll look at a way of storing AES keys using a class called ProtectedData. So let's get started by looking at what symmetric encryption actually is.

What Is Symmetric Encryption?

Symmetric encryption is a two-way encryption process that uses the same key for both encryption and the decryption of your message. In the diagram you can see on the screen, you can see we have plain text of our original data represented on the left, which is then encrypted with a key to produce an encrypted cipher text. Then that data is decrypted using the same key to uncover the original data or plain text. Therefore, this process is referred to as symmetric encryption, as we're using the same key for both encryption and decryption. Let's look at some of the advantages and disadvantages of symmetric encryption. When using a secure algorithm, symmetric key encryption is extremely secure. One of the most widely used symmetric key encryption systems is the US government designated Advanced Encryption Standard. As of writing this course, Advanced Encryption Standard, or AES, as it is also known, is currently unbroken. One of the drawbacks of public key encryption systems like RSA, which we'll explore in the next module, is they need relatively complicated mathematics to work, making them computationally intensive and quite slow. Encrypting and decrypting a symmetric key algorithm is more algorithmic and, therefore, more performant and fast on a computer. The biggest problem with symmetric key encryption is you need to have a way to get the key to a party you are sharing data with. Encryption keys aren't simple strings of text like passwords. They're essentially blocks of random data. As such, you need to have a safe way to get the key to another party. There are ways to use the power of symmetric encryption with a good key sharing scheme, which we'll look at later in this course when we explore hybrid encryption. When somebody gets their hands on a symmetric key, they can decrypt everything encrypted with that key. When you're using symmetric encryption for two-way communications, this means that both sides of the

conversation can get compromised. With asymmetric public key encryption schemes like RSA, someone that gets your private key can decrypt messages sent to you, but they can't decrypt messages that are sent to another party since that is encrypted with a completely different key pair. Before we go into more detail by looking at the Advanced Encryption Standard, let's look at a more historic set of algorithms called DES and Triple DES.

DES and Triple DES

Before we start talking about our preferred symmetric encryption algorithm, which is AES, I first want to talk about DES and Triple DES. I'll start off by saying that in a new system, you never want to use DES or Triple DES, but it's still worth talking about. There's so much older legacy systems may still maintain data that is encrypted with it. The Data Encryption Standard, or DES, as it is known, was developed in the early 1970s at IBM. And later, in 1977, the algorithm was submitted to the National Bureau of Standards to be approved as Federal Information Processing Standard 46, otherwise known as FIPS 46. With consultation of the National Security Agency, the National Bureau of Standards accepted a slightly modified version of DES, which became the FIPS 46 standard. The data is encrypted in DES using 64-bit blocks where the input data to be encrypted is split into 64-bits or 8-byte chunks, which are encrypted using a 56-bit symmetric key to provide confidentiality and privacy. The original DES cipher key size of 56-bits was generally sufficient when the algorithm was designed, but the availability of increasing computational power made brute force attacks more feasible. There are several projects to try and break the DES standard. The most well-known is the DES challenge project. DESCHALL, short for DES challenge, was the first group to publicly break a message, which used the Data Encryption Standard, becoming the $10,000 winner of the first set of DES challenges proposed by the RSA security company in 1997. It was established by a group of computer scientists led by Rocke Verser, assisted by Justin Dolske and Matt Curtin and involved thousands of volunteers who ran software in the background on their own machines connected by the internet. They announced their success on June 18th, only 96 days after the challenge was announced on January 28th. If you want to read more about how this was done, and it is a fascinating story, then I recommend you read the book, Brute Force: Cracking the Data Encryption Standard by Matt Curtin. Once DES had been compromised, there was a race to find something more secure. The answer to that was a modified algorithm called Triple DES, which, as the name suggests, encrypts data three times using DES. The first variation uses three separate 56-bit keys. The data is first encrypted with key 1, then key 2, and then finally key 3. To decrypt the data, the operation is carried out in the reverse direction. First, you decrypt with key 3, then key 2, and then finally, key 1. The second variation only uses two keys. To encrypt the data, you encrypt with key 1, then key 2, and then finally, again, with key 1. And then, again, to decrypt it, you

go in the reverse direction with key 1, then key 2, and then finally key 1. For the purpose of this course, this is all we really need to know about DES and Triple DES. Remember, you shouldn't use this for any new systems. Let me look at some AES code later. I'll quickly show you where you can find some DES and Triple DES example code. DES, Triple DES, and AES, which we'll look at in the next video clip, will show a common programming interface and that they derive from the symmetric algorithm base class. Once you know how to use either of them or the AES implementation, you know how to use all of them. So let's now take a look at the Advanced Encryption Standard.

Advanced Encryption Standard (AES)

We discussed the DES and triple DES algorithms are examples of encryption algorithms that we don't want to use in new and modern systems. So for our discussions around .NET cryptography, that leaves us with AES, which stands for the advanced encryption standard. The advanced encryption standard is the latest encryption standard adopted by NIST in 2001 for the symmetric encryption of messages. The AES algorithm was selected as part of a contest to find a replacement for the data encryption standard. The advanced encryption standard, which is also referenced as Rijndael, which is its original name, is a specifications for encryption of electronic data established by the United States National Institute of Standards and Technology, otherwise known as NIST. The Rijndael Cipher was developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, who submitted a proposal to NIST during the AES competition selection process. Rijndael is a family of ciphers with different key and block sizes. The AES NIST selected three members of the Rijndael family, each with a block size of 128 bits with 3 different key lengths, 128, 192, and 256-bit keys. AES has been adopted by the US government, it's now used worldwide. Like with DES, the algorithm described by AES is a symmetric key algorithm meaning the same key is used for both encrypting and the encryption of the data. In the United States, AES was announced by NIST as the standard FIPS 197 on November 26, 2001. The announcement followed a 5-year standardization process in which 15 competing designs represented and evaluated before the Rijndael Cipher was selected as the most suitable. AES became effective as a federal government standard on May 26, 2002 after approval by the US Secretary of Commerce. AES is available in many different encryption packages and is the first publicly accessible and open cipher approved by the National Security Agency for top secret information when using an NSA approved cryptographic module. For the purpose of this course, it isn't important to understand the inner workings of AES, but for interest, the encryption process is split into several intuitive rounds. These rounds are highlighted in red on the screen. Once I have finished, there is a final round that is performed. These are performed on what is called a block where a block represents a small piece of data that you want to encrypt. You don't need

to encrypt all your data at once, it is split down into smaller pieces and these smaller pieces are what are called the blocks. The number of times each round is repeated, its dependence on the key size. So 10 rounds of repetition is used for 128-bit keys, 12 rounds of repetition is used for 192-bit keys, and finally, 14 rounds of repetition is used for 256-bit keys. Each round consists of several processing steps, each containing four similar, but different stages, including one that depends on the encryption key itself. A set of reverse rounds will apply to transform Ciphertext back into the original plain text using the same encryption key. Round keys are derived for each round from the cipher key using a key schedule. AES requires a separate 128-bit round key block for each round, plus one extra. So how secure is the advanced encryption standard? Governments or businesses place a great deal of faith in the belief that AES is so secure that its security key could never be broken. Also, the key infusion encryption determines a practical feasibility of performing a brute force attack with longer keys exponentially more difficult to crack than shorter ones. A brute force attack involves systematically checking all possible key combinations until the correct key is found and there is one way to attack when it is not possible to take advantage of other weaknesses in an encryption system. The table on the screen now shows a possible number of key combinations with respect to a key size. Highlighted are different key lengths for DES and AES. Notice in the table the exponential increase in possible combinations as the key size increases. DES is part of a symmetric cryptography algorithm with a key size of 56 bits that has been cracked in the past using a brute force attack. There is an argument that 128-bit symmetric encryption keys are computational secure against a brute force attack. Consider the times to crack in this table compared to key sizes. As shown in this table, even if a supercomputer with today's standards, it would take 1 billion billion years to crack the 128-bit AES key using a brute force attack. This is more than the age of the universe, which is 13.75 billion years. Of course, we don't know what the future of computing holds in store for us, but for the moment, AES is a very secure algorithm and it hasn't been compromised by brute force attacking it. Even though 128-bit key should be more than sufficient, it is common to use a 256-bit key in practice. Let's watch our code demo for AES we'll use in a bit. Like when we took a brief look at DES and Triple DES, the AES implementation also inherits from the common abstract base class called SymmetricAlgorithm. This means that once you know how to use one of the algorithms, you know how to use them all as you'll see in the demo. The SymmetricAlgorithm class has some common parameters that we need to populate. The first is the CipherMode. Block cipher algorithm such as AES, DES, and triple DES encrypt in data blocks rather than in single bytes at a time. Block ciphers use the same encryption algorithm for each block. Because of this, a block of plain text will always return the same Ciphertext when encrypted with the same key and algorithm. Because this behavior can be used to crack a cipher, cipher modes introduce that modify the encryption process based on feedback from the earlier block encryptions. The available encryption modes are cipher block chaining, or CBC, Ciphertext feedback, CFB, Ciphertext stealing, CTS,

electronic codebook, ECB, or output feedback mode, OFB. In terms of the overall security of the AES algorithm and its mode of operation, with the AesCryptoServiceProvider implementation in .NET, the best default to use here is cipher block chaining, which is what our examples we use.
AesCryptoServiceProvider is the first implementation of AES that we're going to look at in the next demo, but since .NET core 3.0, there are 2 new modes that are available called CCM and GCM, but these are not available in the AesCryptoServiceProvider code. but we should look a demo of how to use these later on in this module. While CCM and GCM are not new modes to AES, they have only recently just been added to .NET Core. But for now, we're going to look at how to use the cipher block chaining mode. If the project you are working on uses a version of .NET prior to .NET Core 3 and it's perhaps now a legacy system, then it will most likely be using cipher block chaining with AesCryptoServiceProvider, which is what we're going to explore first. The next property we're going to look at is padding. Padding specifies what paddings were applied to a message block being encrypted, which is shorter than the full number of bytes needed for a cryptographic operation. There are different padding schemes included like ANSI X923, ISO 10126, none, PKCS7, or just all zeros. The default padding in .NET for AES, DES, and Triple DES is PKCS7, and unless you have a good need to change it, you should just go with the defaults, which is what we're going to do in our demos. The key property is a byte array that is used to store the encryption key prior to running encrypt and decrypt operations. You can call the RngCryptoServiceProvider object and generate a byte array of the desired key length or you can call the generateKey method on the AesCryptoServiceProvider class. Internally, the generateKey method uses RNGCryptoServiceProvider to generate its key, either way is fine. The final property you need to provide when
using AesCryptoServiceProvider is the initialization vector, or IV property, and this is a byte array that is used to store an arbitrary number that can be used along with your secret key for data encryption. This number, also referred to as a nonce or number once, is employed only one time in any encryption session. The use of an initialization vector prevents repetition and encryption making it more difficult for a hacker who is using a dictionary attack to find patterns to break a cipher. The initialization vector for AesCryptoServiceProvider is 16 bytes in length and needs to be shared with the person decrypting the data. The initialization vector is not considered a secret piece of information, but it is required for successful decryption of that data. In versions of .NET prior to .NET Core 3, there are 2 implementations of AES provided. There is AesManaged
and AesCryptoServiceProvider, but which one should you use? They both provide the same functionality in that they both implement the AES encryption specification, but a main difference is that AesManaged is a .NET specific implementation, whereas, AesCryptoServiceProvider uses the underlying cryptographic libraries in Windows, which are FIP certified. If you want to ensure you are encrypting data with AES by using a compliant implementation, then AesCryptoServiceProvider is the implementation that you'll want to

use, especially if you need to interoperate with systems that are also compliant with FIPS 140-2. This means if you encrypt your data in AesCryptoServiceProvider, then try to decrypt it with another FIPS 140-2 certified platform say in Java, for example, then it is guaranteed to work. AesManaged does not have that guarantee. The code examples in the demo are all based around AesCryptoServiceProvider and not AesManaged, although their usage is very similar apart from their class name. The common language runtime uses a stream-oriented design for cryptography. The core of this design is CryptoStream. Any cryptographic object that implements CryptoStream can be chained together with objects that implement streams. So the streams output from one object can be fed into the input of another object. The intermediate results, the result from the first object, does not need to be stored separately. So let's now take a look at a demo.

Advanced Encryption Standard (AES) Demo

In the demo, we're going to look at using AESCryptoServiceProvider to encrypt and decrypt some data. I will demonstrate how to generate your encryption key as a large random number, which in this case will be a 256-bit key, which is 32 bytes in length. I will also do the same for the initialization vector, which is 128-bit, which is 16 bytes in length. While the focus of this course isn't to use DES or Triple DES, I'll show you where you can find the code demos in our solution, just in case you need to use it. Okay, so what we want to do now is look at how to use AESCryptoServiceProvider to encrypt and decrypt some data. So what we're going to do is we're going to go to our solution, we're going to open folder three, and there's various different projects in here. So what we're going to do is we're going to open up this first project called AES. Now in here, I have two implementations that I've written. So I've got AesEncryption and AesGCMEncryption. We're not going to worry about AesGCM just for the moment, that will be in the next clip. So let's open AesEncryption and see what we have. So I've requested a class called AesEncryption, so everything we need is within this class. Now, just like before, we have our GenerateRandomNumber method, so we pass in a length, in this case it's going to be 32, and then that will create RNGCryptoServiceProvider, initialize an array of the correct size, generate our random number, and then return it. Then we have two methods. We have encrypt and decrypt. So let's look at encrypt first. Apparently we need to encrypt some data that's completely self-contained into this method. So we have three parameters that we're passing in, so we have a byte array of our data to encrypt. So if it's a string, then we need to convert it to a byte array first. We have a byte array of our key, and we have a byte array of our initialization vector. Remember the initialization vector, it's not secret value, but once you've encrypted some data with a specific iv, to decrypt the data you have to provide that same iv. So if you're going to store your encrypted data somewhere, you have to store the iv along with it. So what we do is we instantiate an instance of

our AESCryptoServiceProvider class. Now here I'm explicitly setting the mode and the padding, but these are actually the default, so I'm using cipher block chaining as our mode, and our default padding mode is PKCS7. So then we assign our key and we assign the initialization vector. Now as we previously described, AES uses streams. So what we do is we create a memoryStream, and then we chain that to a cryptoStream. Now you can see here that we call aes.CreateEncryptor that we pass into our cryptoStream. So then we write to our stream, passing in the data to encrypt and its length. Then once you flush that final block and you call ToArray on the memoryStream, you get you get a byte array back, which is your encrypted data. To this point, we've encrypted some data using our 32 byte key and our initialization vector. Now the decrypt method looks very, very similar. In fact, it looks almost identical. So again, we have a method called Decrypt, we pass in our dataToDecrypt, so we're passing in the already encrypted data here. We pass in the same key that we used to encrypt the data, and the same initialization vector. So again, we create an instance of AesCryptoServiceProvider. So we set our mode of cipher block chaining and then we set our default padding of PKCS7. We assign the key and we assign the initialization vector. Then just as before, we create an instance of our memoryStream and then we chain it to a cryptoStream. Now the difference here is instead of calling aes.CreateEncryptor, we're calling aes.CreatesDecryptor, so that's the main difference in this method. So then call write on the cryptoStream whilst passing in our dataToDecrypt. Then we call FlushFinalBlock, convert our memoryStream to an array, let me return the byte array back, which contains our decrypted data. So we're going to want to run this demo, so I'm just going to set it as my startup project. Now let's look at Program.cs, and I've actually got a couple of demos that run in here. So the TestAesGCM we're not actually going to run yet, we're going to save that for the next demo. So what our demo's going to do is we're going to call this TestAesCBC, so I can put the chain in. So we have our text to encrypt. We generate our random number for our key, which is 32 bytes. We generate our initialization vector. We then call encrypt, we then call decrypt, and then we should see that our data that was encrypted is then decrypted back to the original message. So let's run this and see what happens. Okay, so, we have our key that we generated, 32 bytes, just like we've done in earlier modules in the course. We have our initialization vector, which is 16 bytes in length. So now I want to encrypt our data. So we get our text, which was just a text string that says text to encrypt, and I'm going to be calling GetBytes on the UTF8 objects, that's going to convert it into a byte array. So here we can see how we've got 15 bytes of data that we want to encrypt. So we construct the AesCryptoServiceProvider instance, set our mode and padding, assign the key and the initialization vector, create our memoryStream, chain it to a cryptoStream, we call write, which encrypts our data, and then at this point, we have our data encrypted. So this should be a byte array with our encrypted data in it. Now we want to decrypt that same piece of data, so again, construct the AesCryptoServiceProvider object, set the mode, padding key initialization vector, create a memoryStream, chain it to a cryptoStream, but remember, we're calling

aes.CreateDecryptor this time in there. Call write on the cryptoStream, passing in our encrypted data, flush the final block, convert it to a memory array, then we should have our decrypted data here. So what I do is I turn that byte array, the encrypted data back into a string, so it should say text to encrypt. So there you can see that we've encrypted our data and then decrypted it. Okay, and we can see that reflected onto the console screen. We have our original text, our encrypted version of it, and then the decrypted version back into that original text. So that's how to use AES, AesCryptoServiceProvider, more specifically. Now what I want to quickly show you is how to use DES. I'm not going to spend too much time on this, because really, you shouldn't be using DES or Triple DES unless you're in a very old legacy software scenario. But I want to show you it just so you're familiar. Now if we look at the DES class, again, we have our RandomNumberGenerator, exactly the same as what we've just looked at. Now this code here for encrypting with DES looks identical, absolutely identical, apart from one thing. Instead of calling or constructing AesCryptoServiceProvider, we're constructing DESCryptoServiceProvider. But apart from that, the code is absolutely identical, and the same goes for decrypt data. Now in its usage, again, it's virtually the same, but what we're doing here is when we create our key, we're creating 8 bytes. Now, with DES, it only actually uses a 56 byte key, but obviously, the key we're using is slightly larger than that, so the bytes at the end are just padding. Now we also create our initialization vector, 8 bytes in this case, and then the encryption and decryption process is exactly the same as what we've just looked at for AesCryptoServiceProvider. Now again, with TripleDES. So if we look at the help class I've created, again, this is going to look very, very similar, our GenerateRandomNumber method, identical. The encrypt method looks exactly the same except we are creating an instance of TripleDESCryptoServiceProvider. Everything else is identical to what we've just looked at. Now, slight difference here. So, before, I said that there is a two key and a three key variant. So the two key variants we'll encrypt with key one, then key two, and then key one again, and it will do the reverse operation to decrypt our data. So when I generate a key, instead of generating 8 bytes, I'm generating 16 bytes, so you can imagine that is two keys sandwiched together. Now, the implementation in .NET will handle decoupling those keys, you just generate a 16 bytes number, which covers both of those keys. If I wanted to use the three key variation, then I'd comment out this line, and bring this one in, which generates a 24 byte key, which in this case is three versions of the key: key one, key two, and key three. So that's the only real difference, everything else is exactly the same. So, it's worth you having a little play around with it just so you're familiar, but as I said, you don't want to use this in any new systems. But it is quite possible that you have to interact with an older legacy system that does use Triple DES, so in that scenario, you probably want to use this. But if you're generating or creating a new piece of software, then I advise you to steer clear of DES and Triple DES.

AES GCM and CCM Modes

Up until now, we have used AES crypto service provider to provide our AES encryption. In the previous demo, we focused on the cipher block chaining mode. Prior to .NET Core 3.0, this was the best implementation that we had built into the .NET base class libraries. But since .NET Core 3, we have another two modes available to us. CCM mode, which is counter with cipher block chaining and MAC, and GCM, which stands for Galois/Counter Mode. A typical scenario with symmetric encryption, such as cipher block chaining mode, is that you should encrypt your data and then calculate a HMAC of your encrypted data using the same symmetric key that you encrypted with. This is illustrated on the diagram on the screen. Encrypt first, then MAC. .NET Core introduces CCM and GCM to the base class libraries, but these two additional modes of encryption are not new to the world of cryptography, but they are new to .NET. With CCM mode, encryption is performed using cipher block chaining, and the MAC is calculated as part of the encryption process, but the MAC is calculated on the original plain text and not the final cipher text. This is generally regarded as not desirable in most circumstances. With GCM mode, the message authentication is performed on the encrypted cipher text itself instead of the plain text, which is more desirable. GCM ciphers are the most widely-used ciphers used in the world today. With GCM, most implementations perform the authentication check of the MAC and the encryption in parallel, which makes GCM a high performance implementation of the algorithm. If you're in a position to use GCM in your software systems, then you should, if you can. We'll look at a demo of how to use GCM in a moment, but let's look at how it works first. When you encrypt some data, you pass in your plain text as you are used to so far. An example on the screen, it is the text, Mary had a little lamb. You also provide a key, 128, 192 or 256 bits in length. In our demo, we're going to use 256 bits again, which is 32 bytes. We also provide another random number called a nonce, which in this case is 12 bytes. This is like the initialization back to in principle, but it's 12 bytes instead of 16. Like with an initialization vector, this is not a secret value, but you do need the same nonce for decryption as you do for encryption. You can also provide what is known as associated data. This is just data that is passed in with your data to encrypt, and it is stored along with the encrypted data. So, for example, if you're encrypting a customer record, an associated data might be something like their customer number. It's arbitrary associated data that you can provide. It's also optional, so you don't need to provide it. When the GCM encryption process returns, you get your cipher text, but also another value called a tag. This tag is essentially the result of the HMAC that is generated internally to the GCM process. When you go to decrypt your data, you have to pass in the tag, and this will be used to perform an internal validation check on the cipher text to ensure that it hasn't been tampered with. So with that, let's take a look at a demo.

AES GCM Mode Demo

In this demo, we'll take some data and encrypt and decrypt it using AES-GCM. We'll need to generate a new key, as well as the 12 byte nonce value that is passed into the encryption algorithm. So let's take a look. So if we go back to our solution in folder 3, we're going to go back to our AES class. Now previously, we talked about this class here, AesEncryption, and we skipped over the AesGcmEncryption file, so we're going to look at it this time. And you'll be glad to know that the usage of AES and GCM mode is incredibly easy to use. So what we looked at previously for AesCryptoServiceProvider is probably the most complex bit of code we've looked at in this course so far. AES-GCM mode, if you're using .NET Core 3.0 or higher, is actually remarkably easy to use, which is fantastic. So let's look in our class. So again, we have our GenerateRandomNumber method, exactly the same as what we looked at before. No change there. Now let's look at the Encrypt method. So this time, our method takes a byte array, which is our data that we're going to encrypt. We take a key, 32 bytes in this case. We have a byte array of our nonce, which is conceptually similar to the initialization vector, except this time it's 12 bytes. Then we have a byte array, which is our associatedData. Now, this is really, the best way to think of it is just some arbitrary metadata that you can pass in, along with your encryption method. It's not encrypted. It's not secret. It's just if you want to pass some metadata along, you can, and it all gets nicely packaged up with your ciphertext. Okay, so how do we use this? Well, it's very simple. So we generate a pre-instantiated byte array for our tag, which is 16 bytes long. Now, the tag, don't forget, is a HMAC of the data that gets returned to us, and we compute a pre-computed byte array of our ciphertext, which is the same length as our data that we're encrypting. Now to use this, it's very simple. So we get our AesGcm class and we create an instance of that, and we pass the key into the constructor. And then we simply just call aesGcm.Encrypt, pass in the nonce, pass in the dataToEncrypt, pass in a reference to the empty ciphertext array that we want to populate, pass in our empty tag array, and we also pass in the associatedData, which can be null if you want it to be. So once we've done that, our tag and ciphertext are populated. So at that point, I return a tuple of both our ciphertext and our tag. So, very, very straightforward. Decryption is just as simple, so here we have a Decrypt method. We pass in our cipherText byte array, we pass in the same key, we pass in the same nonce. We have our tag, which we previously generated for our encryption process. Then we have a byte array to put our associatedData into. So we create a byte array for the decryptedData. We construct the AesGcm class again whilst passing the key in as a constructor. Then we call aesGcm.Decrypt, and this takes the nonce, the cipherText, a tag, our decryptedData, a byte array that we created, and the associatedData. Okay, so let's take a look at running this as an example. So I'm going to uncomment out this line here that we looked at before. So in our demo, again, we have a string, "Text to encrypt." We construct our aesGCM class. We create our key,

which is 32 bytes. We create our nonce, which is 12 bytes. Then here we call encrypt, which gives us our tuple back, and then we called decrypt. So, I think the best thing to do is just to run this. Okay, so let's step into our method. So we have "Text to encrypt," which is what we're going to encrypt. We construct our aesGCM class. We create our key, 32 bytes, we create our nonce, which is 12 bytes, which we can see here. Now I'm going to encrypt our data. Now because it's a string, we've converted it to a byte array first, which is in here. So let's instantiate our tag, or our empty tag array. Let's instantiate our ciphertext, create the instance of AesGcm, and then call encrypt. Now, at this point, our tag is populated, so that's our HMAC of the data, and our ciphertext is populated, so we return that as a tuple. So now we want to decrypt that data, so that tuple that we returned was put into a variable called results. So here we can pass those individual pieces of data in. So again, we create a pre-instance byte array of our decryptedData, which is empty at this point. Construct AesGcm, pass in the key in as a constructor. Then we call Decrypt by passing in our nonce, the cipherText, a tag, our empty array that we want to populate, and our associatedData. Let me call decrypt because this is our decrypted data. So now we can reflect those onto the screen, so we can actually truly see that our data was decrypted properly. And here we can see that it is. So our encrypted data is that field in the middle there, and then we decrypted it back to the original text. So as you can see, AES-GCM in .NET Core 3.0 and higher is incredibly easy to use. So if you're creating a brand new system and you haven't already got AES encrypted data in there, so you're starting from fresh, then I really do urge you to go to GCM first.

Key Management with Protected Data

Symmetric encryption with AES gives us a lot of benefits when it comes to encrypting data. It is reasonably fast and efficient, and gives us a great level of security. But the handling of encryption keys is a problem. How do you store a key? Do you write the key data to a file? Probably not a good idea, as someone might get hold of that phone. Do you just write the key into a database? Again, this carries risks if your database is compromised in any way. Do you encrypt the data with something else? Well, you most certainly can, but again, how do you manage to case what you're encrypting this key with? There are good ways to do this with asymmetric encryption, which we will cover later in the course. But there's another technique available to you. A useful class to look at is the ProtectedData class. The ProtectedData class gives you a way to encrypt some data using the underlying Windows Data Protection API, also known as DPAPI. The Data Protection API encrypts your data using machine credentials, which means you do not need to maintain or control the keys for your encryption process. To use the Data Protection API, you need to install an additional NuGet package called System.Security.Cryptography.ProtectedData. Unfortunately, even though the code that uses the data protection class will compile on MacOS and Linux, you'll get a platform not supported exception when you try to run

it. ProtectedData is a Windows-only mechanism. As you'll see in the demo in a moment, ProtectedData works by encrypting a byte array of data. This data could be anything, it could be a password that you converted to a byte array or a secret configuration item, or, as we'll demonstrate, an AES encryption key. When you encrypt data with protected data under DPAPI, you'll have to specify what is called a data protection scope, and this exposes two values in an enum. The first is called CurrentUser. With this scope parameter, your data is encrypted at the user level. This means if you encrypt some data with user A and try to decrypt logged in as user B, you'll get an error. This means you need to think carefully about what user you're using, if you use this data protection mechanism to protect encryption keys. Only threads running under the current user context can unprotect the data. The second scope value is called LocalMachine. This ProtectedData is associated with the machine context as opposed to a specific user. Any process running on the computer can unprotect data that was protected on that machine. This enumeration value is usually used in a server-specific application that runs on a server where untrusted users are not normally allowed access. Just take a look at this in the demo.

Key Management with Protected Data Demo

In this demo, I'll show you how the Data Protection class works. I'll first show a simple example of protecting and unprotecting some data. Then I'll show you how to use this with AES, and more specifically, AES GCM. So let's get started. If we go back to the solution file into folder 3, we have a project called ProtectedDataExample. If we open this up, we have a few files in here. So first of all, we have one called AesGCMEncryption. This is exactly the same as the file we looked at in the previous demo, so I'm not going to go over it again. Then we have a class called Protected.cs and this is where we're implementing the use of the Protected data class. So inside my Protected helper class, I have four methods. Now two of them work explicitly with strings, so we've got protect and unprotect, and they take strings and return strings. And then we have another version which takes byte arrays and returns byte arrays. So I'll explain those first. Okay, so in Protect, we take a byte array of a string or a piece of data that we want to protect so this could be an encryption key. We take a byte array of some additional entropy. So a bit like the initialization vector, this is an additional piece of random data which is included as part of the encryption process. Again, just like the initialization vector or the nonce with GCM. This additional entropy is not a secret value, but you do need to store it. And then we have our data protection scope, which is where we can specify whether this is using local machine account to encrypt or a local user. All we simply do is we call the Protected data class and we call the Protect method, which is the static method. So we pass in the data we want to encrypt, our additional entropy, and we pass in the scope, and then that will encrypt our data and return it, it's very simple to use. Conversely, with the Unprotect method, we pass in our encrypted data as a

byte array, we pass in the same entropy that we had during the protect process or null, and we also have our data protection scope, which needs to be the same as what we called protect with. Then we call ProtectedData.Unprotect, we give it our encrypted data, we give it the optionalEntropy, and we give it the scope and then that will decrypt the data. Now, I do have two additional helper methods here, so these are exactly the same, except they pass in strings. So there will be an element of converting strings into byte arrays or converting encrypted strings into Base64 encoded strings. So they're just kind of there for your ease of use. Okay, so let's look at our demo in program.cs, so there are two things that we're going to do here. First, we'll use a quick ProtectedDataTest where we're just going to show how to use protected data. And then I'm going to show how to use it with GCM. So let's run it and step through the code. So let's step into our ProtectedDataTest method. So what we have here is a string, Mary had a little lamb, we have some additional entropy, which is just some random characters that I typed out, and we're using the current user profile here. So if I call into our helper method, first of all, we're converting that string into a byte array, then we're passing in our additional entropy, and we're also passing in our scope, which is set to current user, and we're then getting our encrypted data, which is in this byte array, and we are converting it to a Base64 string. So now, we have our encrypted data here, which we can see as a Base64 encoded string, we can see it's well and truly encrypted. So now we want to take that encrypted data and we want to unprotect it using the Data Protection API. So we pass it into our unprotect method, our helper method. We convert that Base64 string into byte array. We pass in our optional entropy and we'll say set the scope, which again, in this case, is CurrentUser. We can then return our decrypted string. So if we go to the console, we can see we've got our encrypted string that's being reflected onto the screen, and then we can see the results where we've decrypted that string back into our original text, Mary had a little lamb. So what that's done is that's used the Data Protection API through the Protected Data class to encrypt some data at a local user level, which is fantastic so we don't have to manage the keys ourselves. Windows will handle this for us. Okay, so if we carry on the bug in, so we'll have a slightly more complex demo here, so we're going to use this along with GCM. So let's go through this bit by bit. So we have some text which we want to encrypt, and then I create an AES GCM key, which is first 2 bytes. I create our nonce, which is 12 bytes. Now, what I'm going to do is I'm going to call it with helper method that I have here to encrypt some text. So all that's doing is that's just calling our AES GCM helper, exactly what we saw in the demo in the last module or the last clip. Okay, so now if we imagine that we're running maybe a Web API or a web service as some description. So we have encrypted our data, but we want to protect our key. So what we're doing is we're going to generate some entropy. So in this case, I'm creating 16 bytes of entropy and I'm now going to pass in our encryption keys. This is what we used to encrypt our data with GCM. I'm going to pass in our entropy, in this case, we're using CurrentUser as our scope. So I'm passing those in directly as byte arrays where it's in the demo we saw a moment ago, I used strings, but we're using the byte array

versions. So we're going to protect our data and return it. So what we have in here is our encryption key that has been encrypted. Now at this point, we could get that protected key, we could convert it to a Base64 string, and you could just go and store that in a database if you wanted to because it's encrypted. So let's now assume that we've come back into another service method so this might be a few minutes later, and we want to decrypt some text using GCM. So the first thing we're going to do here is we're going to call a decrypt text helper method that I've created. Now what we first have to do is we have to unprotect that key. So if you imagine at this point we don't have any access to the original unencrypted key, so we need to get access to it back. So we're going to use our unprotect method, we're going to pass in the key that's already protected, along with the entropy that we generated for that so that might have been stored in the database as well, and again, we're using our current user scope here. We've gone through and we've called unprotect on the ProtectedData class. So now in here, we have our unencrypted AES GCM key, so that's our original 32-byte key. So now we've done that, we can then call the decrypt method on GCM now that we've recovered that key, so let's just quickly call that. We can get our decrypted text, convert it back into a string, then we can reflect it back onto the screen. So again, as we can see in these few lines here, so the original text said text to encrypt, we encrypted it using AES GCM mode, we then used the Data Protection API to encrypt the original AES key. Now to decrypt the text, we have to then take that encrypted encryption key, unprotect it with the Data Protection API. Now once we've done that, we could then go back and use AES GCM to decrypt our text once you've recovered that key. So this gives us quite a good way of being able to protect encryption keys, which otherwise would be very hard to protect and pass onto other people. We are going to look at another way of doing this using RSA in the next few modules, but for now, this is a really good technique that you can use if you need to store something at a user or a machine level and have it protected by the window subsystems.

Summary

In this module, we first talked about what symmetric encryption is. Then we talked about the DES and Triple DES encryption algorithms. It is not recommended for use in new systems, but it is useful to know about them when you need to work on an older legacy system. Then we looked at the Advanced Encryption Standard, which is a preferred encryption algorithm to use. Next, we talked about two new encryption modes called GCM and CCM that were introduced into .NET with the .NET Core 3.0 release. If you are building a brand-new system, then using GCM is a good idea. Finally, we looked at a way of storing AES keys using the class called ProtectedData. In the next module, we're going to look asymmetric encryption and, more specifically, RSA.

Asymmetric Encryption

Overview

Welcome back to my course, Building Secure Applications with Cryptography in .NET. In this module, we're going to look at asymmetric encryption. First of all, we'll talk about what asymmetric encryption is and its difference to symmetric encryption. We'll then talk about how RSA more specifically works. We'll then look at two different ways of working with RSA in the .NET Base Class Libraries. If you're working on a more legacy code base, then you're most likely to come across the RSACryptoServiceProvider class. We'll also look at the more modern RSA class, which gives you some better options for exporting encrypted RSA private keys. So, let's get started.

What Is Asymmetric Encryption?

The main problem of symmetric encryption is that of securely sharing keys. For a recipient to decrypt a message, they need the same key as the sender. And this exchange of keys can be very hard to do securely. A good solution to this problem is to use asymmetric cryptography, which is also referred to as public key cryptography. With public key cryptography, you have two keys, public key, which anyone can know, and a private key, which only the recipient of a message knows. These keys are mathematically linked. A sender uses the public key of their recipient to encrypt a message, and then the recipient uses their private key to decrypt the message. The term asymmetric is used because this method uses two different linked keys that perform inverse operations from each other. Whereas symmetric key cryptography uses the same key to perform both operations. It is very easy to generate both the public and private key pair, but the power of asymmetric cryptography comes from the fact that it is impossible for a private key to be determined from its corresponding public key. It is only the private key that needs to be kept secret in the key pair. The main advantage of asymmetric encryption is that two parties don't need to have already shared their secret key in order to communicate using encryption. The person encrypting a message only needs to know the recipient's public key, which anyone can know. Then a recipient can decrypt the message with their private key. The main disadvantage is that asymmetric encryption algorithms are comparatively complex when compared to symmetric encryption, which means that the messages take longer to encrypt and decrypt. So let's next take a look at the RSA encryption system, which implements what we've discussed so far about asymmetric encryption.

Exploring RSA

RSA is a public key encryption technology developed by RSA Security LLC, formerly RSA Security, Incorporated. The acronym stands for Rivest, Shamir, and Adleman, the inventors of the technique. The RSA algorithm is based on the fact that there's no efficient way to factor very large prime numbers. Producing an RSA key, therefore, requires an extraordinary amount of computer processing power and time. The RSA algorithm has become the defacto standard for industrial strength encryption, especially for data sent over the internet. RSA is built into many software products. There are some drawbacks to RSA, though. You can only encrypt data that is smaller than the size of the key, so this makes RSA quite limited. It's more common to use RSA to encrypt a randomly generated symmetric AES key. This means that you can then send the RSA encrypted AES key safely to a recipient, and then use AES with that key to encrypt your data. We'll explore this option later in the course when we discuss hybrid encryption. So now that we know what asymmetric encryption is and what RSA is, let's delve a little deeper into how it works. This won't be an exhaustive look at how RSA works, as it can get very complicated and beyond the scope of what this course wishes to achieve, but it is important to have some fundamentals in place to help build up your mental model of the algorithm. RSA typically uses three key sizes. These are 1024-bit keys, 2048-bit keys, and 4096-bit keys. By today's standards, you should use at least a 2048-bit key, as 1024-bit keys have been proven to be weak. Ideally, use the largest key size of 4096 bits, but the larger the key size used, the slower your encryption and decryption operations will be. But this can be negligible with today's computing power. So let's take a look at how we derive our encryption keys and implement the encryption process.

Key Derivation and Encryption

We now know what the RSA key sizes are, but the way they work is very different to symmetric keys that we looked at earlier in the course. Public and private key pairs are based on prime numbers and the difficulty in factorizing prime numbers. What this means is if you have a very large prime number, it's very hard to determine what two prime numbers are multiplied together to make that larger prime. So let's look at a simple example. If I was to pick two prime numbers, 23 and 17, and I ask you what 23 x 17 is, you could quite easily work it out. Give it a try. The answer is 391. You can quite easily work that out on a calculator or in your own head. But if I was to say what two prime numbers do you multiply together to make 5,963, this is much harder to do. Try it and see if you can work it out. So the answer is 67 x 89 = 5,963. Now imagine it's scaled up to numbers much, much larger, like, for example, a 2,048-bit number or a 256-byte number. The strength of RSA keys is based on the fact that going from one huge number back to the two original primes that made that number is a very hard problem to solve. And this is where the strength of RSA comes into effect. Now let's equate this example back to private and public keys. Alice wants to send a message to Bob, so she uses Bob's public key to encrypt the message. In our simple example that we just

went through, 5,963 is a number that is a public key. Alice encrypts a message and sends it to Bob and uses his private key. And in this example, the private key consists of the two prime numbers, 67 and 89. This is a very simplified example. The actual reality is a bit more complicated and it goes out the scope of what I want to achieve in this course, as there are many more components to an RSA key than just prime numbers. But it is beneficial for you to have this simplified mental model of an RSA key in your mind. So as you can see on the screen and what we just mentioned, there are more components to a public and private key. So for example, on the screen, you can see we have P, which is one of our prime numbers, and Q, which is another prime number. So in the example on the screen at the top, we have our public key, in which case P and Q, the two prime numbers, are null because we don't know them in the public key, whereas in the private key we know what the two prime numbers, P and Q, are. Then we also have a Modulus which is the result of both prime numbers multiplied together, as we demonstrated a minute ago. Then we have something called an Exponent, which is a public exponent. InverseQ, and this is the InverseQ coefficient, and these are all present in both the public and private keys, except the InverseQ, which is only in the private key. Then we also have D, and this is a private exponent which is only represented in the private key. Then we have DP and DQ, and these are also exponents that are only present in the private key. But the important bit to note is that within the private key, P and Q are both populated. These are the two prime numbers that you need to calculate the larger prime. And in the public key, P and Q are not present, they are secret to anyone who uses the public key. Unlike with symmetric encryption where the encryption and decryption process is very algorithmic and works by splitting your data into smaller blocks and performing operations on those blocks, RSA takes a more mathematical approach and is based on modular arithmetic. In mathematics, modular arithmetic is a system of arithmetic for integers where numbers wrap around and they reach a certain value. This wraparound value is called the modulus. A familiar use of modular arithmetic is in the 12-hour clock in which the day is divided into two 12-hour periods. If the time is 7 AM now, then 8 hours later it will be 3 PM. Usual addition would suggest that the later time would be 7 past 8, which equals 15. But this is not the answer because the clock time wraps around every 12 hours in a 12-hour clock. Because the hour number starts over as it reaches 12, this arithmetic modulo 12. So let's now take a look at our first implementation of RSA in .NET, which is the RSACryptoServiceProvider class.

Using RSACryptoServiceProvider

Before we move onto a demo of RSACryptoServiceProvider, I want to first talk about a few ways in which you can use this class. RSACryptoServiceProvider is probably one of the most common ways to use RSA in .NET that you'll come across, especially in older code, so we'll talk about that one first. Later in his

module we'll discuss another implementation you can use that has some additional benefits. The first and easiest way to use RSACryptoServiceProvider is to use in-memory objects for keys. This means when we create the RSACryptoServiceProvider objects, we can extract a private and public key into variables. If you wanted to persist a key in any way, then you'll be responsible for saving out each element of the key yourself. Another way to use the RSACryptoServiceProvider implementation is to use a key container built into Windows to persist and load the keys. This is a good way to save out a key and have it be safely persisted on the server that you're executing the code. I'll show you how to do that in the demo. The final technique is exporting keys as XML, which, prior to .NET Core 2, was supported in RSACryptoServiceProvider. From .NET Core 2 onwards, Microsoft decided to deprecate this feature. While it sounds like a good idea on the surface to be able to save out your keys, this led to a lot of bad habits where developers would persist public and private keys to an XML file and then put it onto a server's file system. This is a really bad idea, because if that server is compromised, someone can steal your keys. Sadly, I've worked at two organizations in my career where I found out that private key XML files were stored in a folder called Keys on the server. In the demo in the next clip, I will show you how this works, and I've provided the code if you are maintaining a system older than .NET Core 2. But for anyone using up-to-date libraries, you'll get an exception thrown if you try to save the keys. Later in this module, I'll show you an alternative implementation for RSA that has a way for you to export your keys in an encrypted file, which is a much better solution. But for now, let's look at a demo of using RSACryptoServiceProvider.

RSACryptoServiceProvider Demo

In this demo, I'm going to show you how to use RSACryptoServiceProvider. First, we'll look at how to generate your public and private keys, and I'll demonstrate how to use both in-memory keys and a key container. Then we'll look at how to encrypt and decrypt data with RSA. If we look in our solution file, in the Encryption folder, we have a project called RSA. Now what I want to do first is to look at this file at the bottom called RSAWithRSAParameterKey. Now this is an implementation of RSACryptoServiceProvider that allows us to export our public and private key into member variables in a class. So let's go through this class and see what we have. So we have, so we have two members here, publicKey and privateKey, which are of the type RSAParameters. Then we have a public method called AssignNewKey, and this is going to create our keys. So first of all, I create an instance of the RSACryptoServiceProvider class, and I pass in the size of the key that I want, so in this case, 2048 bits. And then I set a member variable called PersistInCsp or PersistKeyInCsp, which are set to false, because we're not using a CSP for the moment. We'll come onto that in a bit. And then to export my public key, I call rsa.ExportParameters, and I pass in false, and then to export my private

key, I call ExportParameters again, but I pass in true. So by the time this is run, we'll have our public and private keys available to us in those two member variables. So now we have a method that lets us encrypt some data. Now, into this, we pass in a byte array of our data that we want to encrypt, and we return a byte array, which will be our encrypted data. So using this is very straightforward. So we construct our instance of the RSACryptoServiceProvider. We import our publicKey, which remember, we encrypt with the publicKey, and then we simply call rsa.Encrypt, and we pass in the data that we want to encrypt, and we'll return back the byte array of our encrypted data. Decrypting our data is just as straightforward. So we pass in our data that we want to decrypt, and we then create an instance of RSACryptoServiceProvider. This time we import the privateKey, because we decrypted the privateKey. Then we call rsa.Decrypt, and then that returns back our plain text data. So that's an easy example of doing some encryption and decryption. What we also have here is a class that I created called RSAWithCspKey. Now the CSP is a cryptographic service provider, and it's a mechanism provided by Windows, which lets you store your encryption keys in a container inside Windows. So if you're performing these encryption and decryption operations on a server in your data center, then you can have these keys stored in a keystore on that machine. So in this case, we're having a container called MyContainer, and then when we create our key in our AssignNewKey, it's slightly different to what we did before. So we have an instance of a class called CspParameters, and that contains our container name. Then we have a flag that we set, which is UseMachineKeyStore, which does as it says. It's using a machine keystore on the local machine. And then we had to give it a provider name. Now this is quite strange. So you have to give it this big, long magic string. So in this case, it's the Microsoft Strong Cryptographic Provider, and this is the default keystore built in Windows. Now, when you buy hardware security modules, or certainly a lot of the older hardware security modules that you could buy, it would come with a set of drivers that you'd install on the machine, and that would have a different provider name. So if we had a hardware security module by vendor X, it might say something like vendor X, Strong Cryptographic Provider, and then behind the scenes, the cryptographic service provider would know where to store that key. But in this case, we're using the default one built into Windows. So we quit our instance of RSACryptoServiceProvider, we pass in that params class, and then that sets up our keys in that container, or in a container called MyContainer. Now we have a method here called DeleteKeyInCsp, where again, we set up a CspParameters class, and we give it the container name. Then we instantiate RSACryptoServiceProvider with those CSP parameters, and then you can call rsa.Clear, and that will wipe the keys from that container. Now again, we have an EncryptData method, very similar to what we saw before. So we pass in our data to encrypt as a byte array. We create our CspParameters class, and we give it the container name, so it knows where it's going to get those keys from, and it instantiates an instance of RSACryptoServiceProvider, again giving it our key

string and the CspParameters. Now, from that point, we can just call rsa.Encrypt, just as did before. And it's the same deal for decryption. So again we set up CspParameters class where we identify the container name, create the RSACryptoServiceProvider, delete the key strings, and pass in that CspParameters, and then we can decrypt our data. So if we go to program.CS, this new RSA we're going to come onto a little bit later. So I have some examples here. So we're going to show how to use the RsaWithRsaParameterKey, so that's the in-memory keys. Then I'm going to demonstrate how to use the RsaWithCsp. Now it's important to note that this CSP implementation is not supported on macOS or Linux; it's a Windows-only mechanism. So the code will compile on macOS and Linux, but you'll get an exception when you try to run it. So let's run it, and go through this. So I'll step into this method. So we have some data that we want to encrypt called Text to encrypt. We're now going to assign our new key. So we're going to export our publicKey and our privateKey. So if we look in our publicKey, we can see P and Q are empty, and if we look in our privateKey, we can see P and Q are populated. So those are those two prime numbers that we discussed earlier. Okay, so we have created our keys. Now we're going to encrypt our data. So we're going to import our publicKey, and then just call encrypt, which means this should contain our encrypted data. And then again, we can go and decrypt that data. I'll call in rsa.Decrypt. So now notice, we've imported our private key here, because we always decrypt with a private key, and then we can write those details onto the screen, which we can go and look at in the terminal. So there's our original text it wanted to encrypt. We then have our encrypted text below it, and then we have the decrypted and recovered text. Okay, so now we can do the same thing, but this time with the RsaWithCsp. So again, we have our text that we want to encrypt. We're going to create our key. So we set up our CspParameters, and we're going to call our container MyContainer. So at this point, we have our public and private keys in that hidden Windows container. So we can now go and encrypt our data. So we're going to set up another CspParameters, telling it the container name to use. We're going to pass that into RsaCryptoServiceProvider's constructor, and then we can encrypt our data. So this should contain encrypted data. And again, we can go and decrypt that data straight away. So again, we set up the CspParameters, identifying the container name, it constructs the RSACryptoServiceProvider, telling it what container we're using, then we can decrypt our data, and then I can go and clear that key from the container. And obviously, you'd be very careful about doing that in a production environment. Okay, so let's reflect that onto the screen again. Okay, so in our bottom example here, again, we've got our original text, we have our encrypted text, and then we have the decrypted text. So that is how we use RSACryptoServiceProvider, which I'm sure you can see, is nice and straightforward, but let's now take a look at how to do this with the RSA class.

Using the RSA Class

RSACryptoServiceProvider is probably one of the most common ways of using RSA in .NET that you'll come across, especially in older code. There is another class for RSA encryption called, simply, RSA. As we've seen, the mechanism to easily export private keys has been deprecated, but all is not lost if you want to safely back up your private keys in code. In .NET Core 3, two methods were introduced to the RSA class to later to export your private keys in the PKCS #8 private key format. These methods were called ExportsPKCS8PrivateKey and ExportEncryptedPKCS8PrivateKey. What is very helpful for us is that the ExportEncryptedPKCS8PrivateKey method will allow us to password protect the private key before exporting it. By using a password to protect the key, it means you can export the key and store it somewhere safe, such as in a database. Exporting a key is quite easy. First, you set up a PbeParameters instance where you define an encryption algorithm for the password. In this case, it's AES with cipher block chaining. And you'll also tell it what hashing algorithm you want to use. In this case, SHA-256. You can also specify Triple DES, but we are going to stick with AES. You'll also specify an iteration count as the password will be hashed multiple times to slow down the hashing process, much like we discussed earlier in the course. You then call the ExportEncryptedPkcs8PrivateKey method on the RSA class by passing in a byte array of your password and the key parameters that you just created. You don't have to use a human-readable password. You can just create a long, random number with RNGCryptoServiceProvider like we have done in the rest of the course. One question that might be going through your mind from this explanation of exporting keys is how can you protect that password or random number that you've used to encrypt the RSA key? One way in which you can do this is by using the ProtectedData class like we looked at earlier in the course. This means you can encrypt the password at either the user or machine level using the Windows-based Data Protection API. Another option is use a cloud-based key store such as Azure Key Vault, Amazon Key Management Services, or the Google Cloud Key Management Service. As an example, with Azure Key Vault, you can either do all of your RSA encryption using the Key Vault, or you can use the Key Vault Secret Store to store the password that you used to encrypt the private key. Unfortunately, it's beyond the scope of this course to go into detail on Azure Key Vault, but if you would like to know more, then I have some resources for you. First, I have a course here on Pluralsight called Play by Play: Enterprise Data Encryption with Azure Revealed, and this serves as a great introduction to Key Vault. In a second link I have a conference talk that I do called Protecting Encryption Keys with Azure Key Vault. And this goes into quite a lot of detail and contains some development patterns you can use when using the Key Vault. This talk also points to a GitHub repository with lots of code that you can use.

RSA Class Demo

The best way to explain the RSA class is by looking at it more closely in the

demo. First of all, I'll explain how to use the RSA class to create keys and perform a simple encrypt and decrypt operation. Then I'll show the same process again, but this time I'll export the public and encrypted private keys and then reimport them before encrypting and decrypting. So we've looked at how to encrypt with RSA using RSACryptoServiceProvider. Let's now look at the RSA class. So if we open up NewRSA.cs. Now in this class, I have a lot of helper methods to help us out. So we have a private member variable called rsa. And then in our constructor, we call RSA.Create, and we pass in our key string, which, in this case, is 2048 bits. This is going to internally create the keys inside the RSA class for us. I have various helper methods, so I have two versions of encrypt. I have one which takes a string, and then that string is then converted into a byte array inside the method or we have one which just takes a byte array where you can just pass in a row byte array, and then that is encrypted straightaway for us. So to encrypt, we just call rsa.Encrypt. It's very straightforward. And we have an RSAEncryptionPadding mode, which we can specify at the end. Then for Decrypt, I just have one method here, so we have one that takes in a byte array of our data that we want to decrypt. And again, we just call rsa.Decrypt. Now I have a method here, so ExportPrivateKey. This is where we're going to return a byte array of our private key that we can save off somewhere, but it's going to encrypt it for us. So we pass in a number of iterations. So if you passed in 100,000, for example, that means the password is going to be hashed 100,000 times to slow down the process like we discussed earlier in the course. Then we have a string, which is going to be our password. Now it doesn't have to be a traditional password that you might set as a user on a system. So you could, in this instance, generate, say, like a 32-byte or 256-bit random number and then Base64 encode it, and use that big, long password, if you desire. That's typically what I do. So we set up a class here called keyParams, which is of the type PbeParameters. And what we do here is we tell it what encryption algorithm we're going to use. For this instance, we're going to use AES with a 256-bit key in cipher block chaining mode. Then you can also specify what hashing algorithm you're going to use, so our password will be hashed. We have a SHA256, and it will be hashed multiple times according to the number of iterations. So what I do here is I create an array or a new arraySpan of type byte, which is where I'm going to store my encrypted private key. Let me call rsa.TryExportEncryptedPkcs8PrivateKey. I convert my password into a byte array. I pass in the key params and the array span, and then what that's going to do is it's going to store the encryptedPrivateKey in this member variable here, which I can then return to the caller. At that point, you could Base64 encode it and put it into a database, or you could save it into a file. But the important thing is here is that whilst we haven't exported the key, and you do have to do something with it, that key is now fully encrypted using a password that has been hashed, potentially thousands or hundreds of thousands of times. I also have a method here to import an encrypted private key, so if you want to bring an encrypted key back in. So you specify the encrypted key here as a byte array. And again, we have the password that you want to use to decrypt that key. Then we just simply call rsa.ImportEncryptedPkcs8PrivateKey. Convert

that password into a byte array, pass in the encrypted key, and then that will import that key into the RSA class for us. And then finally, we've got the same operation here, but for the public key. But the public key doesn't have to be encrypted, so it's much more straightforward. So we just call rsa.ExportRSAPublicKey, and that gives you that key back as a byte array. Then again, to import it, we have the byte array of that publicKey, and you just call rsa.ImportRSAPublicKey. Specify the key, and it will reimport it for you. So, this class, and it's not very long, it's 57 lines long, and it does a lot for us. It's very, very good implementation. So let's go and have a look at this in practice. So these methods here, we've already tried. So let's just do a straightforward encrypt and decrypt using the RSA class. So we have our text that we want to encrypt. Now, I pass it in as a string in this instance, so it gets converted into a byte array, and we just call rsa.Encrypt, and that gets us back our encrypted data, which we can see in here. Again, to decrypt the data, just pass in the data to decrypt. That simply decrypts it for us. It's very straightforward, which is great. As a developer, I like things to be straightforward. So we can go observe that in the console window. Okay, so in this example, we're going to use an exported private key. So what I do here is create my NewRSA class. Then what I'll do is I'll tell it to export the private key, so I'm going to use 100,000 iterations. I'm using a sort of reasonably complex password, so that password will be hashed 100,000 times before we get the key back. So we've created our key parameters, so we've specified that we want to encrypt our private key with AES 256 in cipher block chaining mode, and we're going to use the SHA256 hashing algorithm for the password. So we tell it to export that key. Now, you notice, it had a little delay as we skipped over that. That's because it was doing that 100,000 hashes first. So this in here is our private key. And then we can also do the same for exporting the public key, which is just a one-line call. Okay, so if you now go into this method here, which is rsa.Encrypt, we're encrypting our data using that public or using that private key just like we looked at in the last demo. So that's our encrypted data. Now if we imagine this was an API call, so we've just called these few lines of code up here, we've encrypted some data, restored it, and then, you know, half an hour later we have another call come in, and we want to decrypt that data. Okay, so we create a new instance of our RSA class. Now I'm going to import my publicKey. So the publicKey is passed in as the byte array. And I'm now going to import my private key. Now again, we had the byte array of our encryptedKey, and we have the original password that we used. So our public key, sorry, our private key has been reimported. And then we can go and decrypt our data that we encrypted earlier. So we have our decrypted data in this byte array. So we can then go and reflect this onto the console window. As we can see here, this is the point where we encrypted our text, but we also exported our public and private key. And then we came back in, and we created a new instance of our RSA class. We reimported both of those keys, including the encrypted private key. We then managed to decrypt our original text. So hopefully you'll agree with me that using the NewRSA class is a much easier and more powerful way of using RSA. And if you're looking after an older legacy system, the chances

are you're probably going to come across RSACryptoServiceProvider. So if you want to actually swap that out for RSA or not is a conversation you had to have with your team, and you need to do some tests to make sure that any old data can easily be decrypted between the two. But if you're building a new system, then I recommend you use this RSA class because it gives you a lot of flexibility in being able to export an encrypted private key, which is quite an important feature to have.

Summary

First of all in this module, we talked about what asymmetric encryption is and its differences to symmetric encryption. Then we talked about how RSA more specifically works. We then looked at two different ways of working with RSA in the .NET Base Class Libraries. If you're working with a more legacy code base, then you're most likely to come across the RSACryptoServiceProvider class. We also looked at a more modern RSA class, which gives you some better options for exporting encrypted RSA keys. In the next module, we're going to take a look at digital signatures.

Digital Signatures

Overview

Welcome back to my course, Building Secure Applications with Cryptography in .NET. In this module, we're going to talk about digital signatures. In this module, we'll take a look at using digital signatures to provide non-repudiation to encrypted messages that you may send to a recipient. We'll first start by looking at what digital signatures are. We'll then look at what the .NET Framework provides to add digital signature support to your applications. We'll first look at using the signature formatters classes to create digital signatures. This will use keys that are generated using the RSACryptoServiceProvider class that we looked at in the previous module. If you work with a lot of older legacy code, then these signature formatter classes are most likely what you'll come across. We'll also look at creating digital signatures using the RSA class that we looked at in the previous module. So let's get started by looking at what digital signatures are used for.

What Is a Digital Signature?

An important function of cryptography is to ensure non-repudiation of a sent message. This is where the receiver of the message cannot deny that the message is authentic. A digital signature is a technique used to help demonstrate

this authenticity of the message. A valid digital signature gives the recipient a reason to believe that the message was created by a known sender such that the sender cannot deny having sent the message. Digital signatures give you both authentication and non-repudiation, authentication because the signature has to be created by using a valid private key and non-repudiation, as the receiver can trust that the message was signed by a known sender, as only they know that private key. So how do digital signatures do all this? Digital signatures are based on asymmetric cryptography. For the receiver of the message, a digital signature allows the receiver to believe the message was sent by the correct sender. This can be thought of as the digital equivalent to a signature on a letter, except a digital signature is much harder to forge. A digital signature consists of the following three parts, the public and private key generation using RSA, assigning algorithm that uses the private key to create the signature, and a signature verification algorithm that uses the public key to test that the signature is authentic. Let's follow through with an example. In this example, Alice has sent a message to Bob, and that message will be signed with a digital signature. First, Alice encrypts some data that she wants to send to Bob. For this example, it doesn't matter if the data is encrypted with a symmetric or asymmetric encryption algorithm. Once this data has been encrypted, Alice takes a hash of that data. She takes a hash of the data because the additional signature is based off RSA, which, as we discussed earlier in the course, has a limit on the amount of data that you can process in one go. If Alice was taking a digital signature for a large file such as a PDF document, then it almost certainly will be too large for the digital signature to process on its own. Hence, why she took a hash of the data first and then created additional signature based off of that hash. Next, Alice signs a data with her private signing key. This creates a digital signature. Then, Alice sends the encrypted data, its hash, and the signature to Bob. First, Bob recalculates the hash of the encrypted data. Bob verifies digital signatures, and the calculated hash, and Alice's public signing key. This will tell Bob if the signature is valid or not. If it is valid, Bob can be confident it was Alice that sent him the message, as it could only have been signed by her private signing key, which only Alice knows. If the signature is not valid, then Bob should not trust the origin and the authenticity of the message. As we just mentioned, the digital signature is based off of RSA. With RSA, when you encrypt some data, you encrypt it with the recipient's public key, and then the recipient decrypts it with their private key. With a digital signature, the signing and verification is the other way around. When the sender signs a message, they use their own private key, and then a recipient verifies the signature using that sender's public key. It's due to the fact that we signed with the sender's private key that a recipient can trust that a message was sent by that sender, as only they will know the private key. So let's take a look at how this is implemented in .NET.

Using the Signature Formatters

There are two methods for generating digital signatures that we'll look at in this module. The first is use the signature formatter classes. If you are working with an existing code base, then it is this method for generating digital signatures that you'll most likely come across. A digital signature requires three main classes to work. The first is RSACryptoServiceProvider. This is needed to manage our encryption keys, just as we have seen in our previous RSA examples in this course. The other classes that are required are RSAPKCS1SignatureFormatter and RSAPKCS1SignatureDeFormatter. These are the classes that we'll use to sign and verify our data, so let's take a look at how this works in a code demo.

Signature Formatter Demo

In this demo, I will show you how to create a digital signature for a piece of data and then show you how to verify if that signature is valid by verifying it. I'll also show you how to make the signature verification fail if the data or the signature is tampered with. In our solution file, if we open up folder 4 called Digital Signatures, then open up the projects in there, which is also called DigitalSignature, we'll have several files in here, so we want to look a DigitalSignature.cs first. Now this is going to be using the RSAPKCS1SignatureFormatter and deformatter. So let's look in the class. So here I have two parameters for my RSA public and private key. I have a method called AssignNewKey. And this uses RSACryptoServiceProvider with a 2048-bit key just like we looked at in the RSA examples in the previous module. Now we export our _publicKey according to ExportParameters(false) and our _privateKey by calling ExportParameters(true). To sign our data, we have a method called SignData, and into that we pass in a byte array of the hash of the data we want to sign because, remember, we're going to take a hash of our data first and then sign that hash. So what we do is we create or construct our RSACryptoServiceProvider object, and we import our private key. So remember, when we sign, we sign with our private key. So then we create an instance of the RSAPKCS1SignatureFormatter, and we pass a RSA object into its constructor, and then we set the hashing algorithm that we're going to use, in this case, it's going to be SHA256. And we do this by passing in a string that says SHA256 in uppercase. And then we just call CreateSignature on the rsaFormatter class, and we pass in the data of the hash that we want to sign, and that will return us a byte array containing that digital signature. Now if we imagine that that digital signature and the hash has been sent to our recipients and they want to verify that the signature is correct. So what we do is we have our VerifySignature method, we pass in the byte array, which is the hash of the data that we want to sign, and we also pass in the bite array of our digital signature. Again, we construct RSACryptoServiceProvider. And this time, we import our public key, so we are verifying the signature using the sender's public key. So we create an instance of RSAPKCS1SignatureDeformatter, again, passing in the RSA object. We set our hash algorithm again, which is SHA256. Then we simply call VerifySignature, and

we pass in the hash of our data, and we pass in the signature. Then that will return true if the signature matches that hash, and it will pass out false or return false if it doesn't match. So if our data or hash or anything has been tampered with or corrupted or maliciously changed, that signature will not match the original hash, and we'll get false returned. So if we go to program.cs. So for this example, we just want this first method. We'll come onto these two later. So in our SignAndVerifyData method, we have our document that we want to sign. In this case, it's just a string that says Document to Sign, and we convert that to byte array. Then we create a SHA256 object by calling the static Create method. Then what we do is we call ComputeHash, and we pass in our documents that we want to sign, and that will give us a 256-bit SHA hash of that document. So now I'm creating our DigitalSignature class, and then I'm assigning a new key. So we simply call SignData, and we pass in that hash, and then that will return us our signature. And then straightaway, I'm calling VerifySignature when I'm passing in the original hash and the signature, and it should return true at that point. And what we can see in the demos if I then rerun it and then change even the signature or the hash, we'll get false returned. So let's run it, and let's do a happy path first. So again, we're creating our document to sign, converting it to a byte array. We're creating our SHA256 objects and then hashing that document. So here we have hash of that document, which is first two bytes in length, and then creating our key. So we just store in the keys in memory for this example. Now I want to sign our data. So first of all, we import the sender's or the signer's private key. Create our signature formatter, set the hashing algorithm to SHA256, and then we pass in the hash that we just created into CreateSignature, and that returns us our digital signature, which we can see here. So now I want to verify that signature. So we import the sender's or the signer's public key. We create our SignatureDeformatter class, set the hash algorithm, and call VerifySignature, passing in both the hash, the original hash that we took, and the digital signature we just created, then that should return true in this instance. We just output these console commands, and look at the console. So there's our original document. This is the additional signature we created, and we can see here that the signature was verified correctly. That's great. So let's just quickly run it once more, and I'm going to step down to the point where we have created our signature. Now what I'm going to do is I'm going to maliciously change the hash. So let's change this 11 to, I'll just change it to 5. So now we have a completely different hash, which doesn't match the original data. So I'm going to verify it again with the sender's public key. If I call verify, in here we can see that it returns false because that signature and hash do not match. So again, if we just look at the console, we can see that the digital signature has not been correctly verified. And that's how to calculate digital signatures using the RSAPKCS1SignatureFormatter and deformatter.

Digital Signatures RSA Class Demo

What I want to do now is show you how to perform the same signing and verifying operations but this time using the RSA class that we explored in the module on asymmetric encryption. This means we get the same benefits of being able to import and export an encrypted private key. First we'll perform a simple sign and verify operation so you can see how it works. Then I'll do the same operation again, but this time by exporting and then reimporting our encrypted private key. If we go back to our digital signature project, we'll see that we have a class here called NewDigitalSignature.cs. So if we open that up, this is going to contain all of the codes the digital signatures using the RSA class. So at the top here we have an instance that we're storing with the class, which I create in the constructor. So in this instance, I'm using a 2048-bit key. Got a little helper method in here just to create a hash, exactly the same as what we looked at in the previous demos. Now here I have a method called SignData. What this does is, it takes our data that we want to sign, so in this case, it's the original document, so I'm not pre-computing the hash; I'm passing the whole document into this method. I then create a hash of that document, and I then call rsa.SignHash. So in this I pass in the data that we want to sign, or the hash of the data we want to sign. I tell it we're using SHA-256, and then I tell it what padding mode to use. And then once we've done that, what we're going to do is, we're actually returning a tuple, so two byte arrays, so we're returning the actual digital signature itself and the original hash of the data. So we're returning the tuple with two items in. So to verify the signature it's very simple. So we have a VerifySignature method. We pass in a byte array of our signature and a byte array of the hash of that original piece of data. For VerifyHash, we pass in the hash of our dataToSign. We pass in the original signature, tell it what hashing algorithm we're using and what padding mode we're using. And that will turn either true or false, depending on whether that signature's valid or not, pretty much exactly the same as what we saw in the last demo except we're using the RSA class instead of the RSAPKCS1 signature formatter and deformatter, so it's a much more straightforward piece of code. Now what I have here is, well, should hopefully look familiar to you. It is exactly the same as what we looked at in the previous demo with RSA. So we have our export private key method. So we pass in a numberOfIterations. That's how many times we want the password to be hashed. We have a string of our password. We then set up our key parameters, and we tell it that our password is going to be encrypted or our private key is going to be encrypted using AES-256 in cipher block chaining mode, using SHA-256 as an internal hashing algorithm, and we pass in our numberOfIterations. And what we do is we call rsa.TryExportEncryptedPkcs8PrivateKey, pass in the password converted to a byte array. Now, again, remember, you can use a 32-byte random number him instead of a human readable password. Pass in the keyParams in the array. And then what that's going to do is, it's going to populate this encrypted private key byte array with our encrypted private key, exactly the same as what we looked at in the RSA module. So again, we have a method to imports our encrypted private key, so we have a byte array of our encrypted key, and we have the original

password. And then we call rsa.ImportEncryptedPkcs8PrivateKey. We give it the byte array of our password and the encrypted key, and that will import private key but decrypt it first. And here we have exactly the same, but for the public key. So we can export the public key. Don't need to encrypt the public key because anyone could know it. It's not secret. And then we have a method to import it. So this is a very useful class to use. It gives us the ability to safely export encrypted private keys and perform digital signatures, so it's really very useful. Let's go back to our demo and we'll quickly show this being used, so we don't want this top demo. What we want is these two here. So what we're going to do is, we're going to run through this demo, and we're going to encrypt., sorry, we're going to sign and verify a piece of data, and then we're going to do the same but with an exported and then reimported private key. So let's just run it and step through it. Okay, so first of all, in our first example, we're going to just sign and verify some data. So we have our document to sign, convert it to a byte array. We create the instance of our NewDigitalSignature class, and we call SignData. So first of all, we compute a hash of our dataToSign, so this is ___ by hash. Then we just call SignHash, which returns a tuple containing our digital signature and the hash, the original piece of data. So now we can go in and verify that piece of data. So again, we pass in our signature and we pass in the hash of the dataToSign. We call VerifyHash and that should return true or false. So in this case, we can see our signature is valid, which is great. So now let's look at our next demo. So this is the sign and verify with key export. This is just showing you how to use the key exporting feature. So here I'm going to create our digital signature class. So then we're going to export our private key. So we pass in the password that we're going to use to encrypt our private key, and we pass in a numberOfIterations. Now this numberOfIterations is the number of times the password's hashed. So similar concept what we discussed earlier in the course. So that's our encrypted private key, and then we also export our public key. Now we have our document to sign. Again, it's just a string that I convert to a byte array. Now, if we imagine that we've exported our public and private key, they've been stored away safely somewhere. We're coming in, and we now want to use them to sign some data. So we're coming in, we have our document that we want to sign. We create our digital signature class. We've created a new instance of it here. So, first of all, I'm going to import the public key. Then I'm going to import the private key, so that would be decrypted first and then imported. Then I'm going to sign some data, so we create the hash of that data first. We then sign the hash, which creates our digital signature, and we can then check that that signature is valued by verifying the hash, and that should say the signature is valued. So that's great. So both those mechanisms work. Again, if I just run this again, generate our signature. Now at the point that we want to verify it, I can go and mess around with either the signature or the hash. So last time we messed around with the hash. This time, let's mess around with the signature. So let's change this 2 to 18. So now we have a digital signature that's not going to match the hash. So if we just step over that line, we can see it says false, so we know that indeed works. The digital signature is not valid. So that shows how to use

digital signatures and signature verification using the RSA class with the added benefit that we can actually import and export our public and private key material safely. So if I was building a brand-new system and I needed to leverage digital signatures, then this is the method that I would favor using.

Summary

In this module, we looked at using digital signatures to provide non-repudiation to messages that you may wish to send to a recipient. We started by looking at what digital signatures are. We then looked at what a .NET Framework provides to add digital signature support to your applications. We also looked at using the signature formatter classes for creating those digital signatures. The keys are generated using the RSACryptoServiceProvider class that we looked at in a previous module. If you're working with lots of old legacy code, then these signature formatter classes are most likely what you're going to come across. We then looked at creating digital signatures using the RSA class, that we also looked at in a previous module. In the next module, we're going to start tying together all of the cryptographic primitives that we've explored in this course to leverage all of their benefits.

Hybrid Encryption

Overview

Welcome back to my course, Building Secure Applications with Cryptography in .NET. In this module, we'll talk about combining a lot of the techniques we have recovered so far in this course to do what is called hybrid cryptography. This is where we use the best of all these primitives to make sure we can encrypt and safely store data for the enterprise. In this module, we'll take a look at hybrid encryption. This will include encrypting data using a combination of both RSA and AES together, followed by using the hash message authentication codes to check the integrity of our encrypted data. We'll also be using digital signatures to ensure that we do not fall foul of any non-repudiation pub by the receiver of a digital signature cannot deny that it was sent by the correct sender. We'll first look at building this hybrid encryption system using some more legacy classes such as AesCryptoServiceProvider, RsaCryptoServiceProvider, and a digital signature formatter classes. Then again, in this module, I'll show you the same example again, but this time I'll use the AES GCM mode and the public key cryptography and digital signatures using the RSA class. Before we look at hybrid encryption in more detail, let's first review some of the security concepts which we are trying to solve.

Reviewing Security Concepts

Back at the beginning of the course, we discussed four security principles that we should tackle with cryptography in our solutions. First, we have confidentiality. Confidentiality is what you traditionally associate with cryptography. This is where you take a message or some other data and encrypt it to make the original data completely unreadable. We looked at AES for symmetric encryption, and RSA for asymmetric encryption. The second type of security operation is integrity. In information security, data integrity means maintaining and assuring the accuracy and consistency of data over its entire lifecycle. This means that data cannot be modified in an unauthorized or undetected manner. Integrity is violated when a message is actively modified in transit. We cover different techniques for given integrity when we looked at MD5 in the secure hash family of algorithms. For example, in this module, I'll be using SHA-256. The third type of security operation is that of non-repudiation. Non-Repudiation is the assurance that someone cannot deny something. Typically, non-repudiation refers to the ability to ensure that a party to a contract or communication cannot deny the authenticity of their signature on a document, or the sending of a message that originated with them. On the internet, a digital signature is used not only to ensure that a message or a document has been electronically signed by the person that purported to sign the document, but also since a digital signature can only be created by one person to ensure that that person cannot later deny that they furnished the signature. Finally, we have authentication. Authentication is where you want to be able to ensure that only an authorized person can perform an operation. Earlier in the module on hashing, we looked at a technique called hash message authentication codes. This is conceptually similar to a hash apart from one thing, the hashing algorithm is also passed a key. This means that if a recipient wants to be able to generate the same hash, they need to possess that same key. If they do not have the key, then they cannot create the same hash. So let's start our exploration into hybrid encryption by combining the best of both worlds from symmetric and asymmetric encryption.

Introducing Hybrid Encryption

So far, we have discussed that for symmetric encryption of algorithms like DES, Triple DES, and AES, that they are far too unofficial when it comes to encrypting data as they're algorithmic by nature, as opposed to mathematical. But a problem with symmetric encryption is that of sharing keys. Sharing symmetric keys such as AES securely between two or more people is very hard to do. Unless you physically hand them the key, moving an AES key across a network is hard to do securely on its own. For asymmetric encryption, natural process of encryption is much slower due to the mathematical nature of RSA, and

there are limits to the amount of data that you can encrypt at once. A real benefits for asymmetric encryption of RSA is how keys are managed. With RSA, you have a public and private key pair. The recipients of the message knows the private key, and they keep that key safe and secret. Their public key can be known by anyone. What we want to do now is get the best of both worlds. We want the fast, efficient encryption properties of AES, coupled with the more secure key mechanism of RSA. What we're going to do is look at what is referred to as hybrid encryption. Hybrid encryption is achieved using unique AES keys, along with symmetric encryption. The public key with RSA is used to encrypt a symmetric AES key. The recipient then uses their private key to decrypt the symmetric key. Once this metric Is recovered, it's then used to decrypt the message that was encrypted originally with AES. Asymmetric encryption can slow down the encryption process, but the simultaneous use of symmetric encryption by forms of encryption are enhanced. Let's run through this step by step with two people, Alice and Bob. First, Alice is going to send a message to Bob using hybrid encryption. Alice generates a 256-bit or first 2 bytes AES key. This key is called a session key. Alice generates 128-bits or 16 bytes initialization vector. The initialization vector is a block of random data that is passed into the AES algorithm to add additional entropy to the process. Remember that the initialization vector doesn't have to be kept secret. Alice then encrypts her message with AES using the session key and the initialization vector. This is the same as any normal encryption process with AES that we've done so far in this course. Alice then encrypts the session key with RSA and Bob's public key. Here we are wrapping the AES session key with the strength of RSA. If anyone wants to decrypt the original message with AES, they will first need to decrypt the RSA encrypted key with the private key. Alice stores encrypted data, the encrypted AES session key, and initialization vector in a separate structure or file. This is the packet of data that is sent to Bob. Now Bob has his packet of information. He wants to decrypt it. To do this, he follows the following process. Bob decrypts the encrypted AES session key by using RSA and Bob's private key. Bob decrypts the encrypted data by using the decrypted AES session key and the initialization vector. Bob can now read the message. If Bob was to send a message back to Alice, then he'll do this exact same process, but in reverse. So he'll start off by generating a new session key, then going for exactly the same process. If you imagine the data being sent between Alice and Bob as a class or package of information, then we have what you can see on the screen, that the RSA encrypted session key, the AES initialization vector, are natural data that was encrypted with AES. Let's take a look at this in some actual code.

Hybrid Encryption Demo

In the following code, I'll show you how to implement what we have just described by using RSA to encrypt a symmetric AES key that is in turn used to encrypt our message to the recipient. In this demo, I'll be using

AESCryptoServiceProvider in cipher block chaining mode, and the RSACryptoServiceProvider. If we go back to our solution and open up folder five, we have various projects in here that we're going to look at in this module. The first one is this project called Hybrid. So I've opened this up. Now in here we have an AES encryption class, I'm not going to go through this again because it's exactly the same as what we looked at earlier in the course, and we also have the RSA with RSA parameter key, again, exactly the same as what we used earlier. Where I'm going to start is with the encrypted packets. So this is the information that's been encrypted and stored, which, if you think back to our conceptual model, was going backwards and forwards between Alice and Bob. So we have our encrypted AES session key, stored as a byte array, our encrypted data, and our initialization vector. In the class HybridEncryption.cs, this is where everything's going to happen. So here we have a method called EncryptData. This takes a byte array of our data that we want to encrypt. I'm also passing a reference to our RSA class, so this is effectively us passing in the RSA keys that we're using. So first of all, we generate our sessionKey, 32-byte random number, 256 bits. This is the key we're going to use with AES. I then create an instance of our encryptedPacket, and I also generate at this point our 16-byte initialization vector, which are just stored straight inside that encryptedPacket. Next, I use AES and I encrypt our data using the sessionKey we just generated, and initialization vector, exactly the same as what we've looked at earlier in the course. And once I've done that and I've installed the encrypted data inside the packet, I then use RSA to encrypt the AES sessionKey, and I then go ahead and store that. So this point we have everything encrypted and ready to send across to our recipient. And in our DecryptData method, we pass in an instance of the encryptedPacket. Let me pass in our instance of the RSA encryption class. Now the first thing that we need to do before we can decrypt our data is we need to recover the AES sessionKey. So the first thing we do is we decrypt that sessionKey with RSA. So this is using the private key to decrypt that key. Then once we have done that, we can go and use AES with our recovered key and the initialization vector that we passed across to decrypt the data and then return it. So here I have some code where I'm demonstrating that, so I'm just going to go straight in and start running it. So first of all, we create our RSA class and generate the key. We construct our hybrid encryption class, and then we go and call the EncryptData method. So we first will generate our sessionKey. Then we create the encryptedPacket and put in the initialization vector, and at this point, using AES, we can encrypt our data using that sessionKey that we just made, and once we've done that, we then want to encrypt our data and store it in the encryptedPacket, which we can see here. So there's our EncryptedData, the encrypted sessionKey, and initialization vector. Now, if we want to go and decrypt that data, first of all, we need to recover the AES session key, because it's encrypted, so we can't use it in its current form. So we do that with RSA and the private key. Let me call aes.Decrypt, using that decrypted sessionKey and initialization vector, which gives us our original data back. So we can then reflect that onto the screen. So here we can see the

message before we encrypted it and then the message after we encrypted it. So that's our first example of hybrid encryption where we're using the benefits of the split key sharing of RSA to encrypt our sessionKey, but the actual encryption of the data itself is encrypted with RSA.

Adding Integrity Checks

Now that we have started our hybrid encryption example, we've now covered the confidentiality security requirement by encrypting our data with AES and the session key with RSA. To further increase the example, we'll add some integrity checking the code. We want to add some integrity checking to ensure that the message that Alice sends to Bob is not corrupted or tampered with in transit. The simplest way to do this is by taking a hash of the encrypted data. This could be done using any of the hashing algorithms, like MD5, SHA-1, or SHA-2. The hash will be calculated after the message has been encrypted and sent to the recipient inside the encrypted packets. Then, when the recipient wants to decrypt the message, I will first recalculate the hash of the encrypted message. If the hashes match, then the data is intact and hasn't been corrupted or tampered with, which means the recipient can safely decrypt the message. As a solution, this works quite well, but we can do better. With the solution of hashing encrypted data, there's nothing stopping the attacker intercepting a message, corrupting the encrypted data, and then recalculating the hash. It would be much better if the hashing of the data could also be protected by the strength of our session key. This is possible by using a Hashed Message Authentication Code, like we discussed earlier in the course. Like a hash code, a HMAC is used to verify the integrity of a message. The benefit we're getting here is that the recipient can only recalculate the same hash if they have the key for that hash. That key just happens to be the session key that will be encrypted with RSA, so they have to be able to decrypt that key first before they can recalculate the HMAC. Let's take a look at one of our previous examples and incorporate the HMAC into it. On the screen, you can see the encryption process we discussed earlier. The only change we're going to make is that we're going to take a HMAC of the data we have encrypted with AES. As a sender, we have access to the AES unencrypted session key, so we will use this as the HMAC key also. Before the sender can calculate the same HMAC, they first have to recover the AES session key by using their private key. If they can't recover the AES session key, then they cannot recalculate the HMAC value. We'd follow the exactly the same process in reverse if Bob were sending a message to Alice. If we were again to visualize what we've just done as a class or packets of data, then we have what you can see on the screen now. In this example, we've added an extra element, which is a HMAC of our encrypted data, along with the initialization vector. This means once the data gets to the recipient, if anyone has tampered with the encrypted data or the initialization vector, we can detect it and disregard the data. This is a very powerful feature, as it gives us a level of security and comfort in detecting any of

our data that has been messed with by a third party.

Adding Integrity Checks Demo

In the following demo, we're going to extend what we previously built and integrate the HMAC, which will be based on the SHA-256 algorithm. Now what we're going to do is we're going to open up the HybridWithIntegrity project file, and you'll see it's going to build on what we had in the last example. So, again, we have our AES encryption, nothing's changed there, RSAWithRSAParameterKey, again, nothing's different. The EncryptedPacket this time has an additional byte array included in there, which is an HMAC validator and initialization vector. And then in our HybridEncryption class, this is exactly the same, except once we have encrypted our session key, what we then do is we create an HMAC. Now, the HMAC is using the unencrypted session key, because we still have access to it. Then what we do is we take a combination of the EncryptedData with the initialization vector appended onto it, and we then compute our HMAC with those two pieces of data. And the reason we include the Iv in there as well, is if whilst this packet is being sent to a recipient, if someone was to change the Iv, the initialization vector, then this hash check on the other end would pick that up. So once we've computed the HMAC, we then store it in our encryptedPacket. Now in decryptData, first of all, it's the same as last time. We decrypt our AES session key with RSA using a private key. And then what we do is we recalculate the HMAC using that session key. So, again, we get the EncryptedData combined with the initialization vector, we compute our HMAC, and we store that. Then what we do is we compare the HMAC that we've just calculated to the one we have sent. If they're different, then we know that something's gone wrong somewhere in between the sender sending the packet and us receiving it, so we abort at that point and we throw a CryptographicException. Something I quickly want to show you is I've got two Compare methods here, and this is quite important to talk about. So if we look at this one here first called CompareUnSecure. Now this is what you traditionally do when you compare two arrays. So first we check the length, and if the length is different, then you return false because the two arrays are obviously different. Then what you do is you iterate through the array and you compare each component or each element of that array to itself. Now, if at any point any of those elements are different, then you'd abort at that point because the arrays are not the same. Now this is quite a fast and efficient way of doing it, this is what you typically do, but in cryptography this is quite unsecure, because the timing of how long it takes to compare that array could be different if they're not equal. So what this means is that an attacker could potentially do what's called a timing attack by observing the timing of the Compare features or the Compare functions. So as I said, what we have is another method here called Compare, and again, it takes two byte arrays. What this does is it will always go through the entire array, no matter whether they're the same or not. So even if the arrays

differ at, say, the second element in the array, it will still go across the entire length of the array during a comparison, which means that you'll get a consistent time every time, or the same time every time, no matter whether the arrays are the same or not, so that's why that was worth mentioning. So once you've compared the HMAC, if they're the same, we can then go and decrypt our AES data. If the HMACs are not the same, then we just don't even bother decrypting the data, because we don't trust it at that point. So let's come in and run this example. So we have our string that we want to encrypt, we construct the HybridEncryption, the class. We create the RSA class and create the RSA keys. We then come in and encrypt our data, so we generate the session key, construct our encryptedPackets. We then encrypt our data with AES using that session key and initialization vector. We then encrypt the sessionKey with RSA and the public key, exactly the same as the last demo. Now we create a HMAC of the EncryptedData with the Iv concatenated to it, and we then store that HMAC in our encryptedPacket. So at this point we have a fully populated packet of data. So now we can go and decrypt it. So, again, we recover our session key with RSA and the private key, we compute a brand-new HMAC of the EncryptedData and the Iv combined, and we then compare it to the one that was sent in the encryptedPacket. So in this case they both match, so that's all good. And we can then go and decrypt our data using AES, because we have now recovered the key. So, again, we can reflect that onto the screen, and we see that the message after decryption is the same as the message before encryption.

Adding Digital Signature

So far, we have a way to securely share an AES encryption key by using RSA as a way to encrypt the AES encryption key with the recipient's public key. Also, we have guaranteed that only the recipient can perform the integrity check by requiring that the AES session key is used to calculate the HMAC. Let's go one step further by adding in the concept of a digital signature. What this means is that the recipient of the message can guarantee who the sender of the message was, which means where recently saved, a message hasn't come from an illicit sender. Let's dive into the details. What you can see on the screen now is the process that we had in the previous video clip with the RSA encrypted session key and the generated HMAC of our data. Now Alice calculates a digital signature using her private signing key and the HMAC that we just calculated. Remember that when we encrypt an RSA, we use a recipient's public key to encrypt, and they use their private key to decrypt. With additional signature, we use the sender's private key to calculate the signature, and the recipient uses their public key to verify the authenticity of the signature. Therefore, we can guarantee that the signature was calculated by the sender as it uses their private key, and only they know it. For this example, Alice stores encrypted data, encrypted AES session key, initialization vector, the HMAC, and a signature in a separate structure, which is then sent over to Bob. You can see this reflected on the data

packet on the screen. The only difference here is that we have added the digital signature of the HMAC. So let's take a look at this in some code.

Adding Digital Signatures Demo

In this demo, I'll show you how to add in a digital signature to our existing code. I'll then show you how to verify the digital signature of the recipient of the data. So let's take a look. In this next demo, we're going to look in the HybridWithIntegrityAndSignature class. And again, everything's the same as the previous demo, apart from a few small bits. So first of all, we have our DigitalSignature.cs class, and this is using the RSAPKCS1SignatureFormatter and Deformatter classes that we looked at earlier in the course. Then we've extended our EncryptedPacket class by including a byte array of our digital signature. And then the main differences here in our HybridEncryption class are that in our EncryptData method we're now passing in an instance to our DigitalSignature class so that means it contains its own set of keys. And then just before create the sessionKey, create the encryptedPacket with the initialization vector, encrypt our data using AES, then encrypt the sessionKey using RSA. And then we create our hmac of the combination of the EncryptedData and the initialization vector. And for the hmac we're using that instance of the sessionKey that we used for AES. And then finally at the bottom here, we create our digitalSignature, and we're creating that digital signature of the Hmac. So the Hmac has already been created of the EncryptedData and the IV, and then we're then signing that Hmac. So then to decrypt, going very similar. So we decrypt our session key using RSA, so it's using that private key to recover that session key. We then recalculate the hmac using that decryptedSessionKey. And then we recalculate the hmac by combining the EncryptedData and the initialization vector, and we then compare that hmac to the one that was sent to us in encryptedPacket. If they both match, then we continue. And at this point, we create a digital, well, we verify the digital signature that was passed in through the encrypted packets along with its corresponding Hmac. If the signature verification returns true, then we are comfortable that it was indeed Alice, in our example earlier, that sent the message to us because only she could have created that digital signature with her private key. And I'm using her public key to verify the signature. Once we're happy at that point, we can then go ahead and decrypt our data with AES. So let's go and run this, so we can see it running. So we create the HybridEncryption, create our RSA class, and create the keys, create a DigitalSignature class and create the keys, encrypt our data. They're running through that hybrid encryption encrypt data method. So we've created the hmac, created the digital signature, now we want to try and decrypt it. So again, we, first of all, check that the hmac is valid. So we recalculate the hmac and compare it with the one that was sent in. In this case, they both match. We then check the digital signature to make sure it's valid. And in this case, it is, which means we can then go and decrypt our data. We can then just reflect that data

onto the screen just to prove that it has decrypted okay. So we have our message after decryption and we compare that to the message before encryption.

Using the AES (GCM) and RSA Class

So far, I've explained how to implement the HybridEncryption system using AesCryptoServiceProvider with cipher block chaining to encrypt our data. And an RSA encryption of the AES key was encrypted using RSACryptoServiceProvider. We've also used the RSAPKCS1SignatureFormatter and the Formatter classes for creating and verifying the signatures. These usages of AES and RSA and digital signatures are very common in the world at the moment. So you're very likely to come across these techniques in existing legacy code. During the course, we've also used the new AES class with GCM mode for symmetric encryption and the RSA class to implement asymmetric encryption and digital signatures. The RSA class gives us the added benefit that we can encrypt and export our private keys safely. Let's now look at the same example of our complete HybridEncryption example but this time using AES in GCM mode and the RSA class for asymmetric encryption and digital signatures. If I'm developing a new system that doesn't contain any encryption already, then this is the way you want to go. So for our final demo, we're going to look in the HybridWithIntegrityAndSignatureGCM project. So, conceptually, the demo that we're going to look at now, so if we look in our EncryptedPacket, so the defaults we have here is we have this Tag, which is the the internal HMAC that's generated as part of the AesGCMEncryption process, which gets returned to us, so we're also storing that Tag in here because we need that to decrypt our data. Then we also have a SignatureHMAC and the digital signature itself. So the SignatureHMAC is the HMAC that we actually sign as opposed to the data. So as in the previous demos, we only had a single HMAC that we calculated. In this instance, we have two. So the Tag is one that's returned to us by AES, and then we had the one that we explicitly calculate ourselves ready for the digital signature process. So let's look in our HybridEncryption class. So I have a little helper method here just for computing the HMAC. And then in our EncryptData method, very similar to what we have before, so we pass in some data that we want to encrypt. We have an instance of our RSA and our sessionKey classes, and they both contain their keys. We generate our AES sessionKey. We create an instance of the EncryptedPacket, and generate this time a 12-byte initialization vector. Remember, in AES-GCM parlance, it's called a nonce, but it's 12 bytes in this case. So, first of all, we encrypt our data with AES-GCM. So for this, we have to pass in the data you want to encrypt, the sessionKey, the 12-byte initialization vector, and that returns us back the cipherText and the tag. And that cipherText and tag we store in the encryptedPacket. We then use RSA to encrypt our sessionKey just like we did before. And then we create our SignatureHMAC by combining both the EncryptedData and initialization vector together. And we create that HMAC using the AES sessionKey. So, conceptually, exactly the same

as what we did in the previous demos. And at that point, we can create a sessionKey, so we are creating that signature of the SignatureHMAC. Now that HMAC is a hash of the EncryptedData and the IV. So at this point, we have our EncryptedPacket, which will contain all of the elements that we need to decrypt that data. So if we then look at our Decrypt method, so first of all, we have to recover that sessionKey. So we're using RSA and the private key to decrypt that AES sessionKey. We then recalculate another instance of the HMAC. So, again, we're combining the EncryptedData with the initialization vector. Once we've done that, we compare that newly created HMAC to the one that was sent to us in the EncryptedPacket. If they match, then we know they haven't been tampered with. So we can then go and verify our digital signature. If that returns true, in that we can verify that signature of the original signature that was calculated and the SignatureHMAC, if that returns true, then we know that it was the correct sender or we actually trust who the sender was that sent us that data. And at that point, we call _aes.Decrypt. So we have to pass in the EncryptedData, the decryptedSessionKey, the 12-bite initialization vector, or nonce, and the Tag that was created during the encryption process. So we parcel all of those in, and then that will give us back our decrypted data. So let's go and run this. So we construct the HybridEncryption class, create our sessionKey and RSA classes so the keys will have been generated at construction time there. So now we want to encrypt our data. So it's this string here we want to encrypt. So we create the sessionKey, create the encryptedPacket and the 12-byte initialization vector, encrypt our message with AES-GCM mode, which we store in the encryptedPacket. We also store the Tag because we need that to decrypt. We then encrypt the sessionKey with RSA. So that has been encrypted with the sender's, sorry, the recipient's public key. We then calculate our SignatureHMAC. So we are calculating that HMAC on a combination of the EncryptedData and initialization vector. And we're using the AES sessionKey as the key for the HMAC. And then, once we've done that, we create a sessionKey of the SignatureHMAC, which we then store in the EncryptedPacket. Now let's decrypt that data. So, first of all, we recover the sessionKey using the recipient's private key. We recalculate a new instance of the HMAC, compare it to the one that was sent in the encryptedPacket. If they match, we can then go and verify the digital signature. So in this case, the signature verifies okay. And then we use AES-GCM mode to decrypt our data passing in the decryptedSessionKey, the 12-bytes, initialization vector, and the Tag. And that should give us our decrypted data back, which we reflect onto the screen. So let's just double-check they are correct. Yep. So we have our message after decryption, which we can then compare to the message before encryption. So exactly the same demo as what we looked at in the previous video. Except this time, we're using the newer AES-GCM mode and the newer versions of RSA and digital signatures through the RSA class. So if I was building a new system and I wanted to build a protocol like this, then this is the version that I would use.

Summary

In this module, we looked at hybrid encryption, where we took some of the concepts we have learned so far in his course, and applied them together to create what is called a hybrid encryption system. This will include encrypting data using a combination of both RSA and AES together. We then followed up by using a hash message authentication code to check the integrity of our encrypted data. We also looked at using digital signatures to ensure that we do not fall foul to any non-repudiation problems, where the receiver of the digital signature cannot deny that it was sent by the correct sender. We initially looked at building this hybrid encryption system using some of the more legacy classes, such as AESCryptoServiceProvider, RSACryptoServiceProvider, and the digital signature formatter classes. I then showed the same example again, but this time I used the AES class with the GCM mode, and a public key cryptography and digital signatures using the RSA class. In the next and final module, we'll summarize what we have learned so far in this course.

Course Summary

Course Summary

Congratulations! You have now completed this course on Building Secure Applications with Cryptography in .NET. We started the course off by looking at the main security requirements that we wanted to solve. These were confidentiality, integrity, non-repudiation, and authentication. Confidentiality is what you traditionally associate with cryptography. This is where you take a message, or some other data, and encrypt it to make the original data completely unreadable. We looked at two solutions for confidentiality; symmetric encryption with AES, and asymmetric encryption with RSA. The second type of security operation was integrity. In information security, data integrity means maintaining and ensuring the accuracy and consistency of data over its entire lifecycle. This means that data cannot be modified in any unauthorized or undetected manner. We looked at various hashing algorithms in this course, such as MD5, SHA-1, and SHA-2, in the 256 bits and 512 bit variants. Most of our examples in the course use the 256 bit variant of SHA-2. The third type of security operation was that of non-repudiation. Non-repudiation is the assurance that someone cannot deny something. Typically, non-repudiation refers to the ability to ensure that the party to a contract or communication cannot deny the authenticity of their signature on a document, or the sending of a message originated with them. We look to implementing digital signatures using the RSA PKCS1 signature formatter and de-formatter classes, as well as using the RSA class for creating digital signatures. The fourth and final security operation is that of authentication. Authentication is where you ensure only the

correct person carries out an action. We explored the use of Hashed Message Authentication Codes, or HMACs, as a way to get authentication. They are the same as hashes, but they also use a key as part of the hashing process. This means to recalculate the same hash, you also need the same key, hence, your authentication aspect.

Further Reading

Now that you've reached the end of this course, I'd like to recommend some books to you if you'd like to learn more. Cryptography is a fascinating subject and it has a really interesting history. The books I'm about to read, I've read personally and I got a lot out of them, so I want to share the list with you. If you're interested in cryptography history, then I recommend this book, The Code Book by Simon Singh. This is a fascinating short book, which is about the size of a standard novel. The book covers the history of cryptography from the ancient times up to the invention of public key systems like RSA. This book is not that technical, it just has a great introduction to the subject. If you like The Code Book by Simon Singh, then this book, The Codebreakers by David Kahn goes into much more detail of the history of cryptography. Be warned, though, this book is huge, it weighs in at around 1200 pages, but it is a fascinating read. Unfortunately, this book doesn't cover modern cryptography, but it does talk a lot about more traditional text-based cryptography algorithms like the breaking of the Enigma code during World War II. The book, Everyday Cryptography Fundamental Principles and Applications, is a more technical look at modern cryptography. This book isn't overly mathematical and it is quite understandable. The book first talks about more of the theory behind some of the algorithms that we touched on in this course. The book then goes on to talk about how cryptography is applied in practical situations like when withdrawing money from an ATM machine. Applied Cryptography by Bruce Schneier is probably one of the most famous books on cryptography. This is an in-depth look at how a lot of modern algorithms work. You do have to have a good stomach for maths though when reading this book, but it is an invaluable resource. I recommend that anyone who likes cryptography get a copy. I just have to thank you for taking the time to watch this course, and I really hope you're taking a lot from it. You're now in a good position to assess if you need to protect any of your employees data, and you have the practical skills to implement it. The hybrid encryption system that we built in this course is a powerful, secure, and great way to efficiently secure your data. I highly recommend you think about putting it into practice. If you'd like to get in touch with me, you can do so via the discussion page for this course. Or you can get in contact with me via my blog's Contact page at www.stephenhaunts.com. I'd also be very grateful if you like this course, so please rate the course on Pluralsight by clicking on one of the star ratings. Now, if you're interested in this subject and any of the other courses that I teach, you can also follow me on my Pluralsight profile, so this means that when I release a new course or update an existing

course, you'll be notified. I really like to engage with the viewers of my course. I'm on Twitter with the name @stephenhaunts. It would be great to connect with you on there. And if you like this course, I'd really appreciate you tweeting about it to let others know about it. Again, thank you very much for taking his course. I hope it has really helped broaden your understanding of using cryptography in your .NET solutions. You have been watching Building Secure Applications with Cryptography in .NET by me, Stephen Haunts. Thank you, and all the best of your continued personal developments.