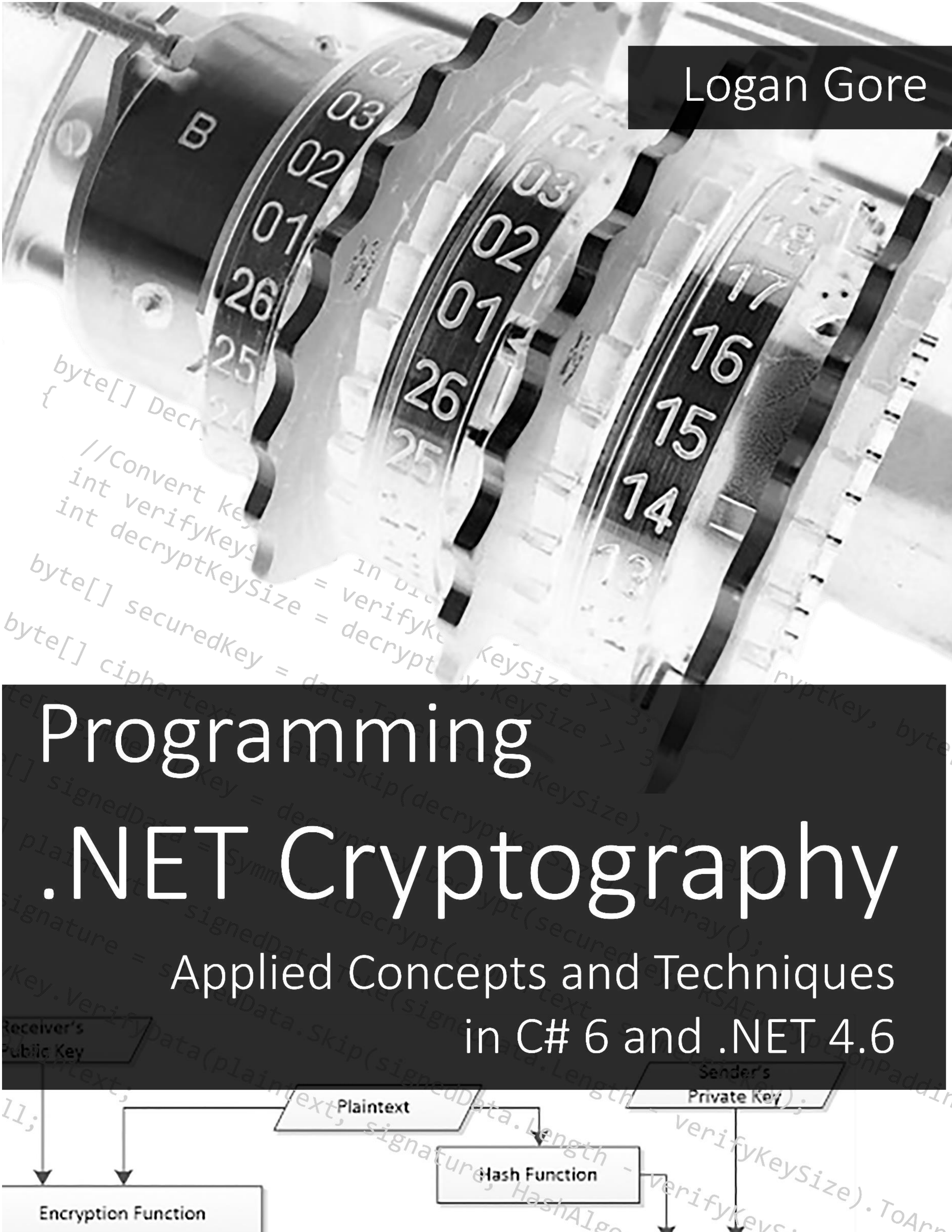
Logan Gore

Programming .NET Cryptography

Applied Concepts and Techniques in C# 6 and .NET 4.6



Programming .NET Cryptography

Applied Concepts and Techniques in C# 6 and .NET 4.6

(This page intentionally left blank)

Programming .NET Cryptography

Applied Concepts and Techniques in C# 6 and .NET 4.6

Logan Gore

Programming .NET Cryptography: Applied Concepts and Techniques in C# 6 and .NET 4.6

Errata and Suggestions

Improvements and corrections will be made to this book iteratively based on reader feedback. Yours is welcomed!

programmingDotNetCrypto@gmail.com

(This page intentionally left blank)

For Mom

(This page intentionally left blank)

Contents at a Glance

(This page intentionally left blank)

Table of Contents

Figures

Tables

Preface

Cryptography plays a critical role in securing assets in our modern world and the world of the future. Historically, cryptography has been a nebulous topic and still is. Most nontechnical individuals have absolutely no comprehension of how cryptography actually works. They have been inundated with misinformation from TV shows and movies. Cryptography has recently taken heat in the media from government officials and political figures describing encryption as a tool used by terrorists. All the while, the rest of the world, including our government, uses this same tool to secure their website logins, bank transactions, personal computers, and files. Cryptography has allowed the Internet and ecommerce to grow by providing a secure means for communication and information transfer. Companies and individuals alike have reasonable assurance that their data and transactions will be protected because of cryptography.

To participate securely in this high-tech world, programming languages and frameworks provide implementations of standardized cryptographic algorithms. The portability and stability of .NET makes it an increasingly popular choice for developing different types of applications. With this demand comes the need to secure these applications and the sensitive data they handle.

*Programming .NET Cryptography: Applied Concepts and Techniques in C# 6 and .NET 4.*6 is designed to serve as a valuable tool for developers who need to use cryptography in their .NET applications. You will learn important security principles and how to program cryptography in the highly popular .NET framework using Microsoft’s flagship language, C#. Most importantly, you will gain an understanding of how cryptography is used to solve commonly encountered security problems in application development.

Audience

This book is for you, the C# programmer who needs to learn how cryptography is used in applications. We do not assume that you have any prior experience or knowledge with cryptography, though it would probably shorten the learning process. You should have experience with C#/.NET. Intermediate-level developers are our target, as we will be using parts of the language with little or no explanation as to their functionality or purpose. For example, we are not going to explain what methods are, how generics work, or how to work with byte arrays. We will explain what is relevant to certain examples and cryptographic programming in .NET. Thus, some common aspects of the C# language will be covered where their functionality has additional implications in our material or examples.

Our Goal

Our goal is for the reader to gain a fundamental understanding of how cryptography and the individual primitives that we will learn how to program can be used to solve problems. We don’t just want you to know, for example, that message authentication should be used; we want you to know *why* it should be used and the specific problem that it solves. We also realize that many readers will use this book as a reference. To accommodate those readers, we've included detailed information in an easy-to-browse format. You will also find helpful tables, lists, and figures.

Organization

This book is organized in a way that teaches you basic techniques and theory and progressively builds on it throughout the book. We do our best to introduce new concepts in a logical manner and avoid the "we'll cover this huge new concept later, but for now let's just use it." There is also a difference between learning the academic side of cryptography and learning how to effectively implement its best practices in a programming language. As a result, we have organized the content to best fit how cryptography is handled in the .NET framework.

We start with an *Overview of Cryptography* and its basic principles. If you are familiar with cryptographic concepts like random number generation, symmetric encryption, asymmetric encryption, cryptographic hash algorithms, message authentication, and digital signing, feel free to skip this chapter. You can always rewind if necessary. Next, you'll move into the concepts behind cryptographic attacks, cryptanalysis and attack methodology. The attacks section will cover some basic models and how they apply to primitives. Again, if you're familiar with these general concepts, you can skip this chapter. Next, we provide an overview of cryptography in .NET. This is an especially helpful chapter for learning the organization of the namespaces and classes that are used for working with cryptographic primitives.

The next several chapters focus on programming with cryptographic primitives in .NET. You will learn how to perform common tasks such as generating hashes, performing symmetric and asymmetric encryption, and digital signing. More importantly, you will learn best practices, practical advice on implementation, advanced concepts, and system design considerations. All of these sections have well developed examples and recommendations for writing your own solutions. This strong foundation will pave the way for learning about slightly more obscure topics that we will cover.

Toward the end of the book we will discuss the data protection API (DPAPI), certificate management in Windows and C#, and general concepts like sound development practices. These later sections are not intended to give you concrete examples; instead, they help you identify common implementation issues and help deliver a fuller understanding of how your solutions relate to a system as a whole.

Chapter Construction

Most chapters are constructed as follows:

Chapter Learning Objectives: These give the reader an idea of what's to come and particular topics or concepts to keep an eye on.

Chapter Introduction: This is a plain-English explanation of the general topic being discussed in the chapter

Chapter Body: The body of the chapter will take a step-by-step approach to explaining relevant concepts as well as how they are used in C#. This will start to taper off a little toward the middle of the book as you'll be familiar by then with many of the cryptographic objects in .NET and common similarities and patterns. Examples appear throughout the chapter to solidify the reader's understanding and explain how concepts can be applied.

Examples: Examples are extremely important in technical books, especially programming texts. Yet much of the literature we see gives poor examples that are hard to follow and difficult to integrate into applications. This leaves developers looking for the clarity these examples are lacking, both conceptually and practically.

This book provides clean examples with full explanation of the concepts and code. Most of the chapter sections will take a step-by-step approach, walking you through the code. A final example will usually wrap everything into a method, or series of methods, designed to provide a fuller understanding aimed at modularity and portability.

It should be noted, however, that our examples do not include what are seen as more production-level programming aspects such as try/catch blocks, logging, or extensive validation; these are often design considerations that depend on the particular solution and scope. That isn't to say that our examples couldn't be used in a production environment. On the contrary, many of the examples have been pulled *directly* from enterprise-level solutions that provided try/catches, error logging, and validation around this code at a higher level. We just leave this aspect of implementation up to you.

Chapter Summary: A summary of the chapter outlining the key areas.

Chapter Questions and Exercises: These questions are designed to challenge the reader to apply the information covered in the chapter. Exercises are designed to reinforce concepts through simple but applicable tasks.

Scenarios: Scenarios are designed to stress critical thinking skills in real-world situations. These are modeled around questions we have found on Internet forums and message boards that involve higher level decisions than simply writing a program; many involve team, project, and compatibility issues.

Text Formatting

Paragraph text is written in Calibri 10pt. Code examples are written in `Consolas 9.5 with a grey shading`. C#/.NET objects are **bold** in paragraphs. References to chapter titles, sections, examples, and methods are *italicized*.

Acknowledgements

I would like to thank my technical research team for all of their hard work and diligent review. I am very grateful for the discussions with my friends and coworkers who were especially helpful in shaping the content and examples. I owe so much to my family for their continued support throughout this process.

Recommended Reading

The following texts are recommended for those of you who want a deeper understanding of cryptography and some of the key concepts in this book:

Writing Secure Code, 2nd Ed. **Michael Howard and David LeBlanc. Microsoft Press. 2003.** An excellent text about secure coding practices. It's getting a little dated but still provides probably the best strategies for secure application coding with tons of Windows specific insight.

***Cryptography Engineering: Design Principles and Practical Applications*. Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. Wiley Publishing Inc. 2010.** This book is highly recommended for its practical advice in system design and applicability to programming cryptography. It explains common mistakes in implementation for different types of algorithms and cryptographic schemes.

Introduction to Modern Cryptography, 2nd Ed. **Jonathan Katz and Yehuda Lindell. CRC Press. 2015.** This goes very deep into the details of algorithms and the mathematics behind various cryptographic primitives. It places a strong emphasis on formal definitions and principles. While it doesn't focus on system design as much as *Cryptography Engineering*, it has huge amounts of practical advice.

Secrets and Lies: Digital Security in a Networked World. **Bruce Schneier. Wiley Publishing Inc. 2004.** This is a good read for anyone in security, especially those who want a higher-level view of tech security in general. However, the first two titles are better suited if you are looking for more detailed information on cryptography.

1 Overview of Cryptography

Cryptography*: The use of mathematical techniques to provide security services, such as confidentiality, data integrity, entity authentication, and data origin authentication.*

Chapter Objectives

1. Identify different cryptographic primitives and their uses.
2. Understand why cryptographic key management is needed.
3. Compare different models of cryptographic attacks.

This chapter is intended to act as a refresher for developers who have prior experience with cryptography, and give a crash course for those who don't.

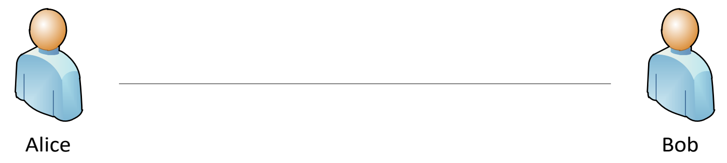
What is Cryptography?

Throughout the ages we've wanted the ability to keep certain communications private. The need for secrecy is recognized by many different groups: governments, militaries, businesses, and individuals. Cryptography is the science or practice of encrypting data.

Historically, the best advances in cryptography have stemmed from war. Communications during wartime often have to be transmitted across public communication channels for efficiency. And depending on how you look at it, basically every communication channel is susceptible to eavesdropping, and therefore public. What followed was a serious need for a system to transmit the secret data in a coded form. Enter cryptography.

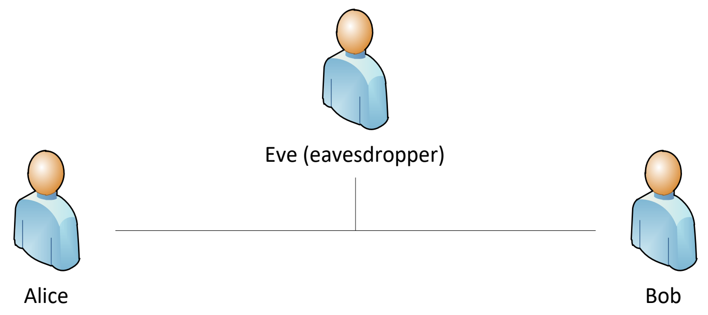
Communication between two people, Alice and Bob, has been commonly used in cryptography texts as an example of communications over a public channel. Figure 1 illustrates this relationship:

Figure 1: Alice and Bob commuicate over a public channel



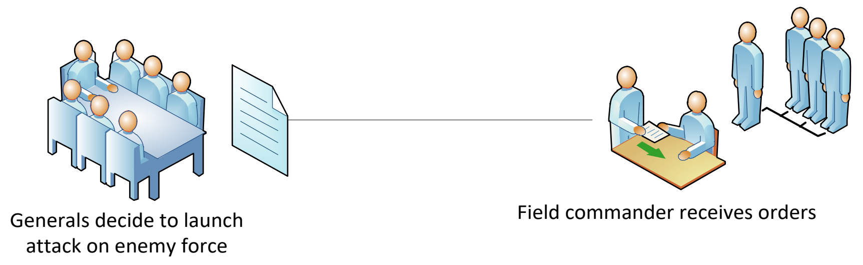
Because this is a public channel we always assume that Eve, the eavesdropper, can hear what is said between them (see Figure 2). Think of this type of eavesdropping as being able to listen to, or view, any communication between them. This includes phone calls, physical messages and mail, email, text messages, and any other type of communication that uses the public channel.

Figure 2: Eve can intercept communications between Alice and Bob



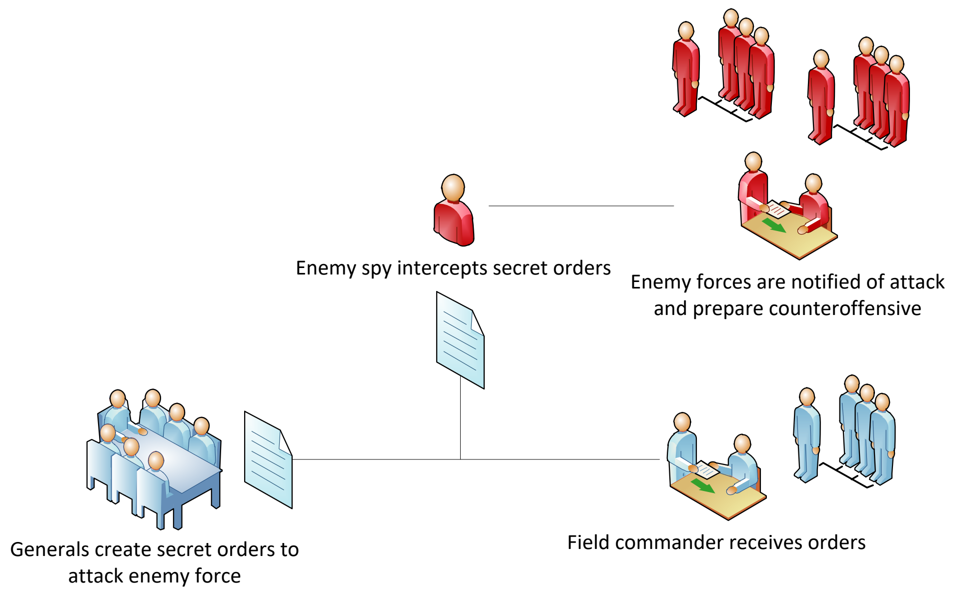
A less boring example of this type of communication model is seen on the battlefield, which is illustrated in Figure 3. Orders from a basecamp or headquarters are handed down to field commanders that instruct them to attack an enemy force.

Figure 3: Passing critical message


Generals decide to launch attack on enemy force
Field commander receives orders

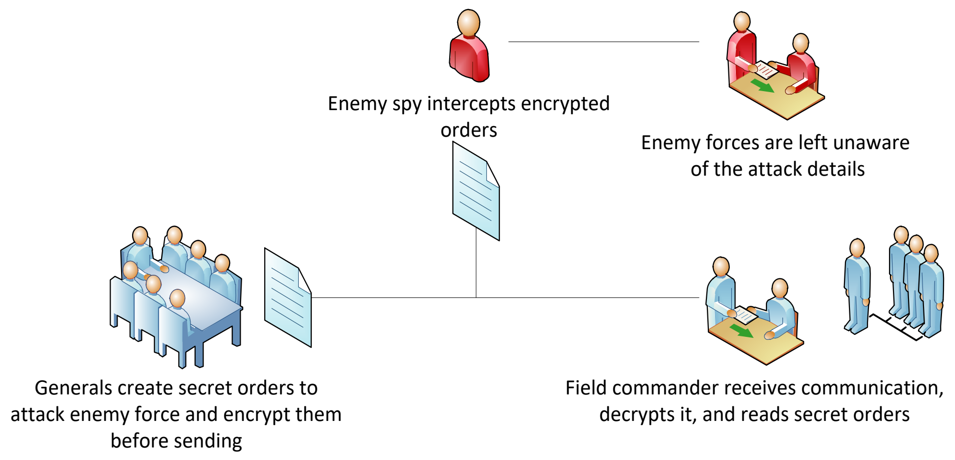
The secrecy of this message is vital to the operation. The attacking force could easily lose the element of surprise if their orders were intercepted by the enemy. Figure 4 shows an enemy spy intercepting the order and notifying the enemy force.

Figure 4: Message intercepted


Enemy spy intercepts secret orders
Enemy forces are notified of attack and prepare counteroffensive
Generals create secret orders to attack enemy force
Field commander receives orders

How can the generals and the field commanders communicate without the enemy spy reading their communications? They need to encrypt their communications. The generals will encrypt the orders before they send them, and the field commanders will decrypt them upon receipt. This type of model, shown in Figure 5, protects the data in transit. The threat of the spy intercepting the communication still remains; however, we can assume that the enemy force cannot decipher the orders.

Figure 5: Encrypted message intercepted


Enemy spy intercepts encrypted orders
Enemy forces are left unaware of the attack details
Generals create secret orders to attack enemy force and encrypt them before sending
Field commander receives communication, decrypts it, and reads secret orders

Cryptographic Primitives

Cryptographic primitives are basic—primitive—parts of cryptographic systems: symmetric encryption, public-key encryption, hash algorithms, MACs, digital signatures, and random number generators. Primitives are used by themselves or in conjunction with other primitives to create secure solutions to problems. Standardized secure protocols and processes use primitives to achieve reasonably secure environments.

It’s important to have a solid understanding of the purpose and capabilities of primitives. This will help you find the appropriate tool in your cryptographer’s toolbox to solve a particular problem. Moreover, a firm understanding will help you identify misuse and improper application of primitives in other people’s systems as well as your own. Table 1 shows how primitives can be used to provide common aspects of security.

Table 1: Applications for Cryptographic Primitives

Primitive	Application
Asymmetric (public key) Encryption	Confidentiality
Symmetric Encryption	Confidentiality
Hash Algorithms	Integrity
Digital Signatures	Integrity, tamper evidence, nonrepudiation
Message Authentication Codes (MACs)	Integrity, tamper evidence
Random Number Generators	Generate secure random data (key material, IV, token).

Randomness

Randomness plays a huge role in cryptography and most cryptographic primitives. Most academic cryptography texts will cover it extensively. We will not get very deep into randomness here. We will, however, cover the key concepts that are relevant to this book.

Entropy: The Measure of Randomness

Sometimes developers need a little entropy in their lives and their code to keep people guessing. *Entropy* means different things to different people. Mathematicians and physicists will often have different definitions. When we talk about entropy in this book we will be using it as a way to *describe* the measure of randomness in data. High entropy would mean that the data is highly random. Low entropy would mean that the data is not very random, and more predictable. In terms of passwords, for example, high entropy 8 character password would look something like “ilyHjNqC”, while low entropy would look like “password” (assuming both are constrained to case-insensitive alpha).

In practical usage we assume that the attackers do not know the target data. So, whether data is high-entropy or low-entropy, the attacker can only try to guess what it is. Obviously, the higher-entropy data will be more random, less predictable, and harder for the attacker to guess. This is what we want. However, under the circumstances where the attacker knows the password, its entropy is not a factor because it is known. So for our uses, entropy is only relevant to unknown data.

As you may have figured out, data with a fixed-size can only have so many possible values. For example, a four-digit PIN number can only have 10,000 possible values. Even if we have a truly random four-digit number, this doesn’t offer us much protection in terms of an attacker trying to guess it. While the attacker doesn’t know the number, they know that it’s between 0000 and 9999. This is why longer and more complex keys and passwords offer much better protection where they are randomly generated.

Random Number Generators

A random number generator (RNG) is a cryptographic primitive that creates a random number or *bit string* (a binary string). Most RNGs allow you to specify the length of this number. RNGs are extremely important in cryptographic solutions. The security that many cryptographic primitives provide is dependent on random data. For instance, the strength of the encryption that a symmetric algorithm can provide hinges on the unpredictability of its key. The more random a key is, the harder it is for an attacker to guess.

However, not all RNGs are made equal. Some RNGs generate what appears to be random data, but is actually predictable. This presents a huge problem. Developers using this type of data expose themselves to huge risks of compromise where attackers are able to exploit the predictability of the RNGs. Well known exploits have stemmed from bad random number generators. A particularly notable example was the Netscape Internet browser RNG vulnerability, which allowed attackers to compromise encrypted communications. Random number generators that artificially generate randomness are called *pseudo-random number generators*.

The obvious question arises: “how do we *generate* data that is unpredictable?” Finding a source for this data has been extremely challenging. The nature of computing makes it relatively easy to determine how, where, and when the machine will create or *find* random data. Some sources such as network traffic, key strokes, mouse tracking, and clocks, have been used as values to add additional randomness to a generator. When an RNG gets enough random input data it is considered to be unpredictable to the extent that it is computationally infeasible to establish an exploitable pattern. This is called a *Cryptographically Secure Pseudo-Random Number Generator*, or *CSPRNG*.

The Frequency Stability Property

Frequency stability can help us determine if data itself is actually random based on how often variations in its composition occur. To illustrate this concept we’ll use a scenario. Our scenario starts by assuming that two people make a decision at the same time every day according to the result of a coin toss (watch TV if heads, workout if tails). The first person actually flips a coin to make his decision. The second person, however, doesn’t use a coin and tries to simulate randomness by guessing (basically just deciding if today is heads or tails). The results are recorded over time in a simple format like binary: one for heads, zero for tails.

Here are our two sets of data:

Set 1: 111000101100011010110001011000111100110101011010111101100010001000000111

Set 2: 101110101101011010001101110101101011010101101110101101011010001101000101

The question is whether we can look at the data and determine which is being generated by the actual coin toss. And the answer is that we can; this is how.

Rather than trying to look at specific patterns, we need to examine the properties of a sequence. Simply measuring the number of ones or zeros (heads or tails) is not sufficient because over time they will be fairly even. We need to use something called the Frequency Stability Property that will expose tendencies of humans to favor certain patterns. The Frequency Stability Property says that a truly random sequence will be equally likely to contain every possible sequence of a given length. Therefore, we look at every possible sequence of three bits, which are:

000, 001, 010, 011, 100, 101, 110, 111

With this we could parse our results on every third bit and see how often the above sequences have occurred. Theoretically, the result with much greater disparity among sequences will be human-generated.

Here's the first set of data:

111 000 101 100 011 010 110 001 011 000 111 100 110 101 011 010 111 101 100 010 001 000 000 111

Table 2 contains the frequency distribution for the first set of data.

Table 2: Frequency distribution for the first data

Sequence	Frequency (number of occurrences)
000	4
001	2
010	3
011	3
100	3
101	3
110	2
111	4

Now, we'll look at the second set, and determine the frequency of the same sequences:

101 110 101 101 011 010 001 101 110 101 101 011 010 101 101 110 101 101 011 010 001 101 000 101

Table 3 contains the frequency distribution for the second set of data.

Table 3: Frequency distribution for the second data

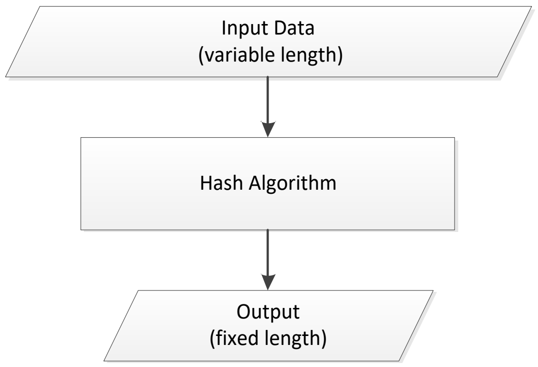
Sequence	Frequency (number of occurrences)
000	1
001	2
010	3
011	3
100	0
101	12
110	2
111	0

The first set of data has a pretty even frequency stability for the 3-bit sequences (given how it was parsed). The second set, being parsed in the same manner, has huge disparity in the frequency of the sequences. Theoretically, these results expose the second set as the simulated randomness.

Hash Algorithms

Hash algorithms are one way functions. They take a variable-length input and render a fixed-length output—the hash. This process is outlined in Figure 6. The hash, also known as a message digest or fingerprint, is an irreversible product of the input. The nature of this relationship makes it computationally infeasible to derive the original input from the hash.

Figure 6: The process of hashing data



Popular hash algorithms include the MD and SHA series. The last addition to the MD series was MD5. While still in wide use, MD5 is no longer considered to be suitable for use in modern production security systems.

A *cryptographically secure* hash function should provide a random mapping between inputs and outputs. As a result there should be no distinguishable patterns visible in the hashes given certain aspects of the input data.

To understand the security of hash functions we have to understand collisions. *A collision* is where two different inputs produce the same hash. Because a hash function can only generate a finite number of hashes (2^n, where *n* is the hash length in bits), and there are an infinite number of inputs, collisions are guaranteed. So why would we use these functions? Because while collisions cannot be ruled out, we can make the probability of hitting one extremely low. We do this by using longer hash functions that allow for more possible outputs, as well as making sure that the algorithms themselves are constructed in a manner that minimize collisions.

Secure hash functions are constructed in a manner that disallows shortcuts or ways to compromise the algorithm itself. If a hash algorithm is considered cryptographically secure, its strength (how much security it can provide) is usually measured by its hash length. The longer the hash, the fewer chances for collisions. For instance, if a secure hash function has a length of 160 bits, it would be considered more secure than a 128-bit hash function. Generally, we consider a hash function's collision resistance to be $2^{n/2}$, where n is the output length in bits. MD5 for example, which has an output length of 128 bits, is only seen as providing 64 bits of security because collisions can be expected at 2^{64}. This relationship between collisions and fixed-length outputs presents an attack avenue known as a *birthday attack*, which will be covered later this chapter.

Symmetric Encryption

Symmetric encryption is characterized by the use of the same key (a secret piece of data, such as a password) for both encryption and decryption. When two parties wish to communicate with each other using a symmetric algorithm, such as the popular Advanced Encryption Standard (AES), they both must possess the key to encrypt outgoing messages and decrypt the incoming. Figure 7 and Figure 8 show the symmetric encryption and decryption models, respectively.

Figure 7: Encryption with a symmetric algorithm

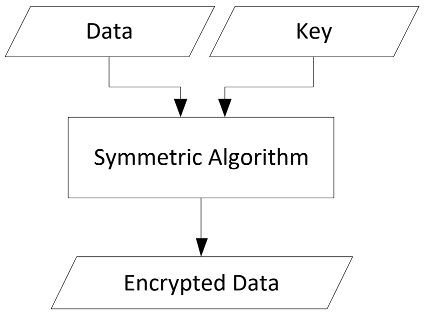
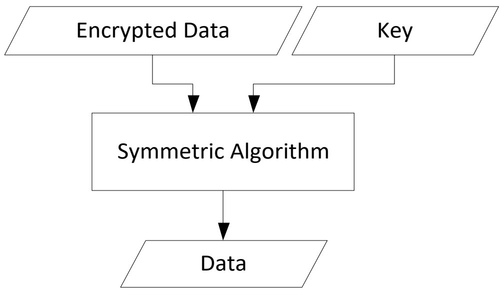
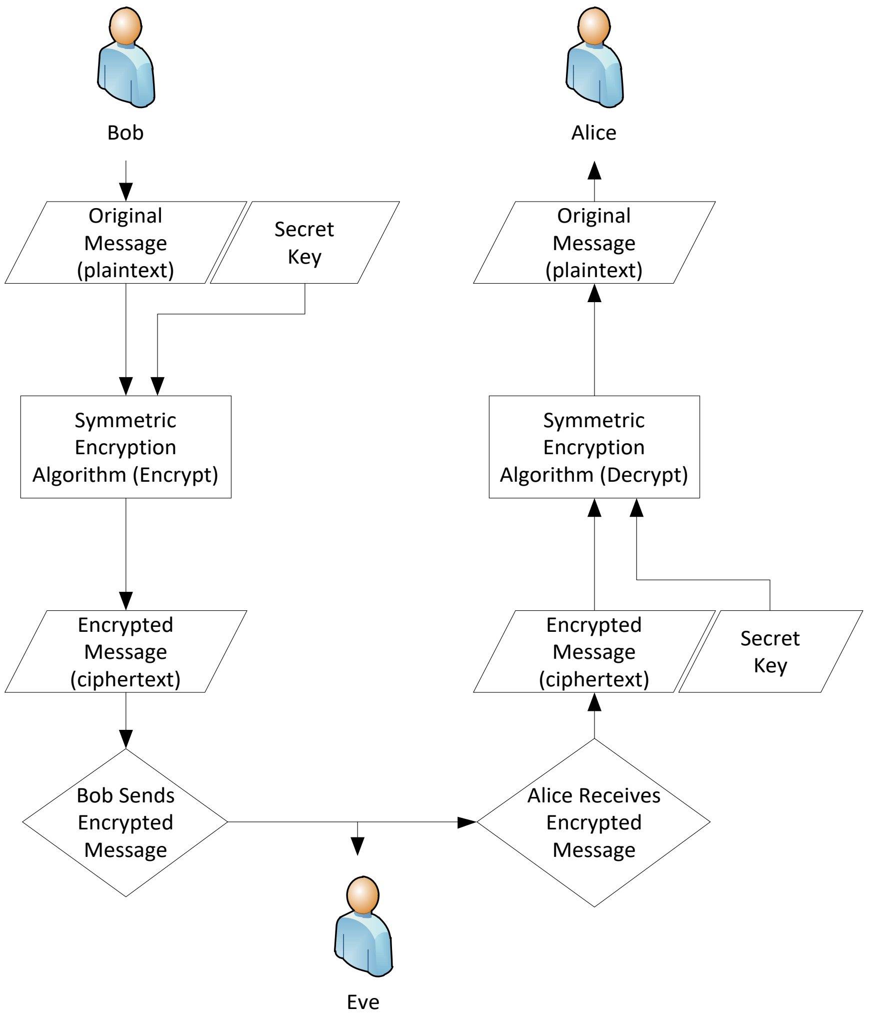


Figure 8: Decryption with a symmetric algorithm



Symmetric encryption secures communications within an otherwise insecure channel. Alice and Bob can send their private messages without worrying that an eavesdropper, Eve, could read their messages. Eve can read the encrypted messages but cannot decipher them without the secret key, which she doesn't have. Figure 9 shows how Bob and Alice communicate using a symmetric encryption system.

Figure 9: Bob sends Alice encrypted data


Bob
Original Message (plaintext)
Secret Key
Symmetric Encryption Algorithm (Encrypt)
Encrypted Message (ciphertext)
Bob Sends Encrypted Message
Eve
Alice Receives Encrypted Message
Encrypted Message (ciphertext)
Secret Key
Symmetric Encryption Algorithm (Decrypt)
Original Message (plaintext)
Alice

But a huge problem is inherent in the normal use of symmetric encryption: how do you securely distribute the keys? For this to work without even seeing the keys, Bob and Alice must already have a copy of the same key!

Alice and Bob can only send and receive secret messages across an insecure public channel if they have already negotiated a key. To be secure, they can't just send it across the channel in plaintext. Using a strictly symmetric key system requires that keys get to the intended parties safely; this could be accomplished face-to-face, or through other *out-of-band* means (a communication channel separate from the primary channel). In networked solutions, the key distribution problem is solved by using another type of encryption, known as *asymmetric encryption*, to securely distribute the keys. Asymmetric encryption will be covered later in the chapter.

Advantages

Besides security in general, the biggest selling point of symmetric key encryption is performance. Symmetric algorithms are extremely efficient (fast) and usually engineered for optimization on hardware.

Disadvantages

Secure key management is the biggest hassle and vulnerability associated with symmetric key systems. How is a secret key securely distributed to the parties that need it? Where is the key stored? Attackers trying to compromise data encrypted by a strong algorithm like AES will have the most luck going after the keys rather than trying to attack the algorithm itself. The chore of managing secret keys is what makes it the most vulnerable aspect of the system.

Because the same key must be distributed among all participating parties, a particular key cannot be easily linked to a particular party. Therefore, it is easier for a party to *repudiate* their involvement in a particular transaction. Later, we will see how this problem is solved using asymmetric algorithms.

Types of Symmetric Algorithms: Block Ciphers and Stream Ciphers

The two main types of symmetric algorithms are *block ciphers* and *stream ciphers*. Both adhere to the basic symmetric model using the same key to encrypt and decrypt. The difference between the two is how the data is handled during the cryptographic operations.

Block ciphers handle data in fixed-size blocks. The cryptographic operation, whether it’s encryption or decryption, is performed on a single block of data at a time. Today the most common block size is 128-bit. These block ciphers process 128 bits of data at a time. A 1280-bit piece of data could be evenly processed as ten 128-bit blocks. Block ciphers have one main disadvantage in that they must have the data on hand, i.e. in memory. This is unacceptable for solutions that must handle streamed data.

Stream ciphers are typically faster than block ciphers and can handle encryption and decryption “on the fly.” As a result, stream ciphers are ideal for solutions that need to encrypt or decrypt streaming data. Some solutions cannot afford to store the data in memory until enough accumulates to process with a block cipher. In these situations a stream cipher is able to encrypt and decrypt data where the data length might be indeterminate and needs to be processed at a high speed.

One-Time Pads

One-time pads (OTPs) are the only truly unbreakable method of encryption. OTPs do not use a regular fixed-length encryption key to secure their data. Instead, they use a randomly generated key that is the same length as the data itself. Regular symmetric encryption can have weaknesses that stem from information leakage, a condition that occurs because a relatively small key has to secure a large piece of data, which can produce patterns in the ciphertext. The more information that a key encrypts, and the smaller the key is relative to the data, the more leakage there is (in simplest terms). An OTP avoids this by only ever using a key once and generating a key that matches the plaintext size. The result is a perfectly secure encryption model.

Message Authentication Codes (MACs)

Keyed hash algorithms, also known as message authentication codes (MACs), add an additional property to the functionality of a cryptographic hash function: a secret key. Whereas a regular hash function produces a hash based on a piece of data, a MAC produces a hash based on the data and a key. Changing a key would render a different hash using the same piece of data. MACs are used in scenarios where you want control over who or what is able to hash or verify data.

There are different types of MAC functions. You could build your own by prepending a key to data being used as input to a hash function such as SHA256. You could also build a custom block-cipher MAC function (we’ll cover these shortly). However, “rolling your own” is not advised because it’s generally not as productive or secure as using a standardized MAC implementation from a cryptographic library. Ultimately, the purpose of a keyed hash algorithm is to make the process of either producing or verifying a hash dependent on the key as well as the data, a process which can pertain to secured or unsecured data.

Using MACs with Secured Data

Encryption gives us a way to communicate privately between participating parties in an otherwise insecure channel. But being that this is an insecure channel, how do we guarantee that an attacker hasn’t tampered with our data while it was in transit? Remember, tampering doesn’t have to successfully compromise an object. It could also be used to disrupt or break a communication channel. With regular encryption we have no method of checking to see if the message was damaged; even worse, if an attacker compromised an encryption key, they could read as well as change messages. In most contexts an attack that changes data is much more dangerous than one that seeks to just read the data. What cryptographic primitives do we have to check data integrity? What about a hash algorithm?

A normal cryptographic hash algorithm produces a fixed-size fingerprint from a piece of data with an arbitrary length. Data that is transmitted or stored in persistent storage usually has its hash attached (prepended or appended) so that it can easily be verified. In the verification process anyone who has the piece of data in question (and its hash) can verify its integrity. It’s this characteristic of hash algorithms that makes them insufficient for message security. If just anyone can verify a piece of data with using hash, than anyone who has access to the data can change the data as they please, rehash it, and attach the new hash. Even though the data has been tampered with, nobody knows they’re looking at bad data because the hash is correct. Under the circumstances of transmitting encrypted data, message authentication codes (MACs) make sure that this sensitive data has not been tampered with, and that only the key holder can attach the correct MAC.

Using MACs with Unsecured Data

MACs can also be used for unsecured or unencrypted data to ensure that it hasn’t been tampered with by anyone who doesn’t have access to the algorithm key. This data doesn’t need to be secured as it’s not necessarily *private*. It does, however, need to be correct and maintain its integrity. MACs provide a means to let only key holders create or verify a message, even in plaintext form.

Asymmetric Encryption

Asymmetric encryption is characterized by the use of two keys: one for encryption and one for decryption. This *key pair* is made up of a *public key* and a *private key.* Each key pair is different but mathematically related. Data encrypted with the public key can only be decrypted with the corresponding private key. As the names imply, the *public key* is publicly known and can be transmitted over insecure channels. The *private key* must be kept secret. In Figure 10, Alice transmits her public key to Bob and Eve observes the transaction. Figure 11 shows a one-way encrypted communication from Bob (encrypting) to Alice (decrypting) using an asymmetric system.

Figure 10: Bob gets Alice's public key

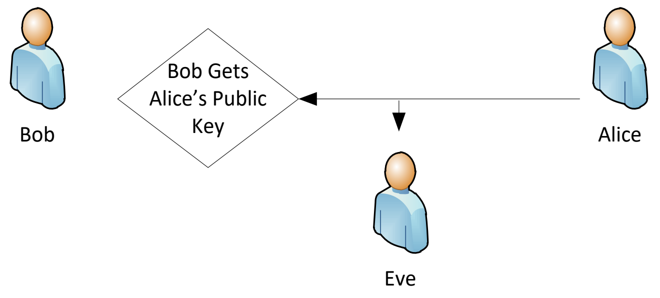
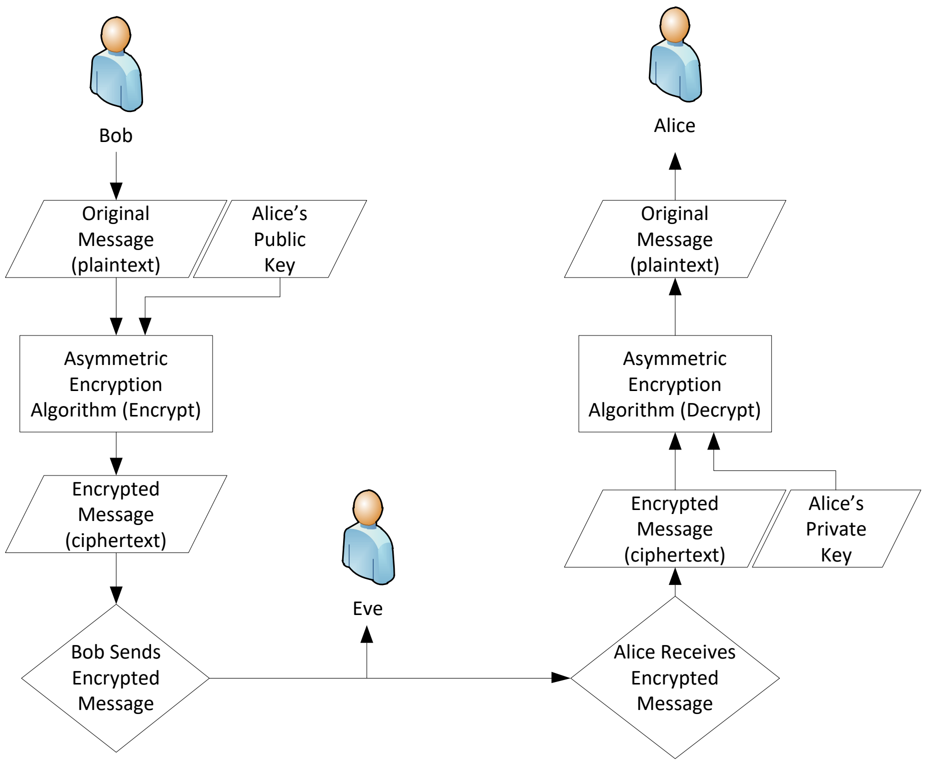


Figure 11: Bob encrypts a message and sends it to Alice


Bob
Original Message (plaintext)
Alice's Public Key
Asymmetric Encryption Algorithm (Encrypt)
Encrypted Message (ciphertext)
Bob Sends Encrypted Message
Eve
Alice
Original Message (plaintext)
Asymmetric Encryption Algorithm (Decrypt)
Encrypted Message (ciphertext)
Alice's Private Key
Alice Receives Encrypted Message

A crucial characteristic of asymmetric encryption is that given one key in a pair, the other key cannot be determined. Therefore, an attacker that intercepts an individual's public key cannot easily determine the corresponding private key. This work factor is based on the size of the key.

Advantages

Asymmetric models solve the secure key distribution issue endemic to symmetric encryption schemes. This makes the asymmetric model the primary method of negotiating symmetric keys between endpoints. Secure protocols like SSL and TLS use asymmetric encryption to provide a secure means of exchanging a shared secret key for a symmetric algorithm. Once the symmetric key negotiation is completed, a symmetric algorithm like AES can be used to communicate securely between the endpoints.

Nonrepudiation is another characteristic that the asymmetric model offers. Nonrepudiation means that a party cannot deny performing an action. This is achieved by each party having their own public and private keys that identify them and an assurance that these keys are kept safe.

Disadvantages

Asymmetric encryption is more computationally intensive than symmetric encryption. CPUs have to work harder and longer. This is why most secure protocols use asymmetric encryption only for negotiating a symmetric key and switch to using a symmetric algorithm once the negotiation has completed. Asymmetric algorithms are also not used for encrypting *bulk data*. Bulk data refers to anything larger than the key size. Symmetric encryption should be used for bulk data. We will cover models for doing this later in the book and how to build a hybrid system using both symmetric and asymmetric algorithms.

Digital Signatures

Digital signatures are made possible through asymmetric algorithms. When using an asymmetric key pair to provide *privacy*, usually the data is encrypted with a public key and the data is decrypted with the private key. A digital signature uses the private key from the key pair to provide *integrity*. It works like this: a hash is taken of the data. Most commonly this is the plaintext, however, some models will sign a ciphertext. The hash is then signed (encrypted) using the signer's private key, producing the *digital signature*. This *digital signature* accompanies the data so that it can be verified. Digital signature generation is shown in Figure 12.

Verification is performed using the *signer's public key*, which decrypts the signature (exposing the hash) and compares the received hash with a hash of the data in question, as computed by the receiver (this is the process taken by most cryptographic libraries). The verification process is shown in Figure 13

Figure 12: Typical digital signature generation in a cryptographic library

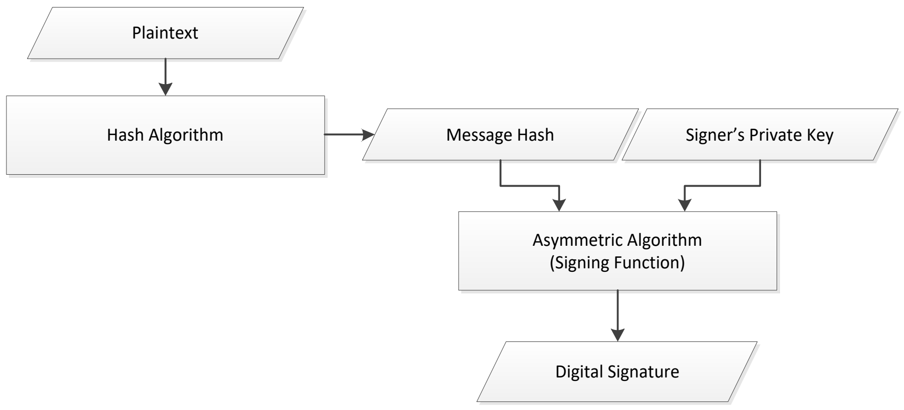
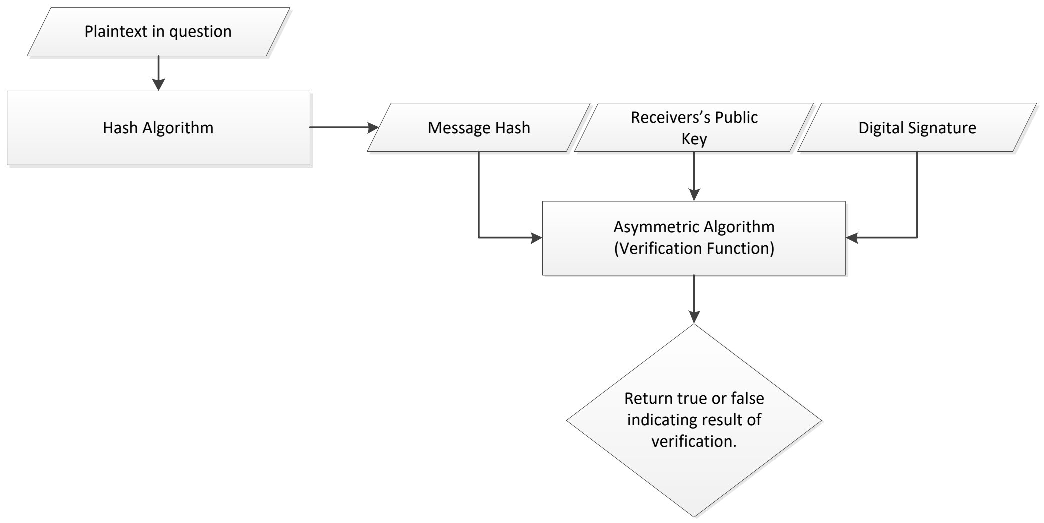


Figure 13: Typical digital signature verification in a cryptographic library


Plaintext in question
Hash Algorithm
Message Hash
Receivers's Public Key
Digital Signature
Asymmetric Algorithm (Verification Function)
Return true or false indicating result of verification.

In the context of secured communication, data will be encrypted with the *recipient's* public key. The signature will be computed with the *sender's* private key. The resulting security is such that only the recipient can decrypt the data, and only the sender can sign the data to where it can be successfully verified with the corresponding public key. The asymmetric model provides a means to enforce something called *nonrepudiation*. Under this concept a party cannot *repudiate* their involvement in an action. As an example, Alice has a key pair that was issued to her at her job. She is expected to digitally sign all of her documents and emails. Because her public key is known and on file she will be unable to deny having signed these documents. Ultimately, this process only links her to a document through her signature, it doesn't guarantee that someone else didn't steal her private key or reuse one of her messages. However, other mechanisms are in place to make sure that this process of trust and key distribution is in place and secure.

Digital Certificates and PKI

The hardest aspects of securing communications in a virtualized world are knowing who we're talking to and if we can trust them. Secure transport protocols like SSL really don't even matter if you can't trust—or don't really know—with whom you're communicating. In the real world we are comfortable giving our money or credit card to the local grocery store because we trust they probably won't do anything scandalous. They have a business license and tax records that could also corroborate that they are in fact ABC Grocery Store. But we don't ever check these things because we can see them first hand. We believe them mostly because they have an establishment. On the Internet people tend to conduct themselves the same way. This can be problematic. The problem is two-fold. First, how do we verify that someone is who they say they are? Second, even if we know who they are, how do we trust them?

Digital certificates give us a format by which we can verify identities and the information associated with them. In the real world most people carry around a driver's license to prove that they are in fact allowed to operate a motor vehicle. It also acts as a basic form of identification. The basic information on the license can be used to verify that the possessor is actually the person on the card: picture, race, age, height, weight, and in most cases a signature. Like a driver's license, a digital certificate contains basic information that identifies the entity: name, domain, their asymmetric public key. Now, although we can see this information, how do we trust it? How do we trust that a driver's license is not a fake?

In the real world, standardized methods of identification typically have features that allow us to *trust* that the ID is actually real. Some features provide more trust than others. For example most driver's licenses have a hologram that is hard to fake unless a professional printer is used. Still, most people just look at it overall: does it look real? It's probably safe to say this will suffice at the local gas station when you're carded buying a very *classy* box of wine. It'll also work when the clerk notices that the back of your credit card says "Check ID" and they stare at your driver's license with that glazed over look.

Some contexts call for an increased level of trust. If you are pulled over by a police officer, she is probably going to verify that the info on your license matches your general appearance and that you resemble your picture. However, she will also need an increased level of trust that the license itself is actually valid, rather than just assessing how it *looks*. For this she will have to turn to a *trusted third-party* who can confirm the license and the information on it: the Department of Motor Vehicles (DMV), or someone else who issues the licenses. When she checks your license against the DMV's records, she will be able to verify that the license itself is real along with your information. She can also check the validity of the license with the DMV since information pertaining to its revoking or suspension probably won't be reflected on your physical license.

The same model is used in the cyber world to verify identities and their status with a trusted third-party. It's called Public Key Infrastructure (PKI). A Certificate Authority (CA) issues digital certificates to people or companies who can prove their identity to the CA. It digitally signs each certificate that it issues. Other people can use the CA's public key to verify the signature on the certificate. However, people must trust the CA in order for this process to work properly. Most popular browsers come with keys from the well-known CA's. When you initiate a secure connection (HTTPS) with a website, your browser will check to see whether the website's certificate was signed with one of the trusted keys you already have. If it wasn't, it will prompt you and ask if you want to proceed without being able to verify the site's identity (this is very risky!).

The PKI model can also be implemented within businesses or organizations. In these models a server will be used as the CA to sign and distribute certificates. Most server operating systems will come with some type of PKI. Windows Server is commonly used and has an easy to configure CA.

X.509

Digital certificates have standardized formats for the data they contain. X.509 certificates are a standard for certificate formatting. X.509 is currently on version three (X.509v3), and basically all of the certs you see will be X.509v3.

Cryptographic Key Management

Key management refers to how cryptographic keys are managed throughout their lifetime. Key management is often overlooked in solution design despite its importance, leading to insecure applications. In fact, it's fairly common to find keys and passwords hardcoded into an application. Developers who are new to cryptography rarely consider the issue of where their keys are being stored, or the amount of time necessary to devise a successful key management strategy. Key management is partially sidestepped in many simple application designs by deriving keys from user input. This technique, however, is not scalable or manageable. Applications demanding higher security and manageability require extensible solutions that can handle key management at scale with the necessary administration. For this reason, key management is perhaps the most difficult aspect of cryptography and one that is not usually covered in most texts.

Cryptographic key management systems (CKMS) are designed to handle all aspects of secure key management. Generation, storage, access/distribution, and archiving are major functions of these systems. To securely perform these functions CKMSs are constructed using basic cryptographic primitives: symmetric encryption, asymmetric encryption, random number generators, MACs, and digital signatures. The type of system that is needed depends on the environment and the requisite level of management.

Common Responsibilities of a CKMS include:

- Key generation
- Controlling access to keys
- Securing keys in storage
- Tracking the usage of keys
- Protecting key chains, or hierarchies of keys
- Maintaining the status of keys
- Archiving keys
- Retiring keys that reach the end of their appropriate usage or lifetime
- Revoking keys that have been compromised

Cryptoperiods

A *cryptoperiod* refers to the amount of time or usage to which a key can be subjected before it needs to be retired. Cryptoperiods vary based on the type of key and its application. Cryptoperiods should always be a design consideration in a CKMS or secure application. Just because data protected with a strong key and algorithm is considered to be secure until the year 2030, doesn't mean that the key should be used until 2030. So, make sure you research the recommended cryptoperiods for your keys given what they are being used for. We will make recommendations on cryptoperiods throughout the book. A good resource for determining safe cryptoperiods for your keys is the NIST Special Publication 800-131A, available at http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf.

Implementation

Implementing a custom cryptographic key management system is beyond the scope of this book and must be tailored to the application and use case. Additionally, key management is generally considered an administrative task rather than one that is handled by programmers. That isn't to say that programmers don't build CKMSs; they do. But CKMSs are most often configured and operated by a security or systems administrator.

CKMSs, however, do not need to be custom developed for all applications. PKI models are popular and offered in many server operating systems. While these do not usually provide an extensive feature or policy set, they can be useful for creating and managing keys for clients.

Cloud based key management is also becoming popular. Here, key management takes place remotely as a service. Clients can access keys from the service, which enforces the CKMS policies that determine how keys are handled and who can access them. One particular issue with cloud CKMSs arises when an organization must maintain physical control over the keys, i.e. store the keys on their own hardware.

This book will not cover key management. We recommend that you use an existing off-the-shelf CKMS or key management service, as building your own is difficult for even most skilled developers.

Secure Protocols

Secure protocols use cryptography to provide a standardized means of securing communications. Common secure protocols include SSH, SSL/TLS, and IPSec. Behind the scenes these protocols use combinations of cryptographic primitives to achieve the desired security characteristics. Having a knowledge of existing protocols is important. Knowing the protections offered by common protocols allows you to assess the security of systems and applications that use them. Additionally, you will save money and benefit from stronger security where you can use an existing protocol instead of writing your own.

Cryptanalysis and Attacks

Cryptanalysis is the science or practice of breaking cryptographic systems and algorithms. Cryptanalysis is very important. Standardized algorithms, many of which we'll be using in this book, have undergone considerable cryptanalysis. RSA for example, has been around for decades. This level of scrutiny is obviously a good thing. It lets us know that talented cryptanalysts have punished the algorithms and not found any significant weaknesses. Still, some attacks are designed to exploit poor implementation or algorithms or system design. You want to have an understanding of basic attack methodology before starting to build systems of your own. This will help you identify particular details of a design that might be susceptible to attack.

In this section we will learn about some of the basic concepts and principles behind attacking cryptographic systems and algorithms. Feel free to conduct additional research for greater detail or context specific examples.

The First Assumption: Kerckhoff's Principle

Before we get into talking about different attack models, we need to cover a very important principle in cryptography: Kerckhoff's Principle. Under this principle we assume that an attacker knows the underlying details of our algorithm, but not the secret input that we give it (usually a key). So, for example, if you are using AES256 to encrypt data that is getting stored in a remote database, you should assume that the attacker knows that you are using AES256.

Ciphertext-Only Attacks

The ciphertext-only attack is the most basic attack and is what most people think of when they imagine "cracking" encrypted data. In a ciphertext-only attack, the attacker only has access to the ciphertext. Remember, the ciphertext is the encrypted data. In this model the attacker is trying to gain information about the plaintext or the key. The manner in which the attack is carried out depends on the algorithm type.

Known-Plaintext Attacks

A known-plaintext attack is where an attacker knows the corresponding ciphertext of a plaintext encrypted under a particular unknown key. In this model the attacker tries to capture pairs of ciphertexts and their *known* plaintexts. The attacker looks for some type of discernable pattern or leakage that could give them info about the underlying plaintexts for other ciphertexts they intercept or the keys used to encrypt them.

Chosen-Plaintext Attacks

A chosen-plaintext attack (CPA) is almost identical to a known-plaintext attack, but in this model an attacker knows the corresponding ciphertext of a plaintext of their choosing. Rather than having to capture plaintext/ciphertext pairs the attacker can generate ciphertexts for whatever data they want.

Chosen-Ciphertext Attacks

The chosen-ciphertext attack (CCA) model gives the attacker the ability to encrypt plaintexts of their choice as well as decrypt ciphertexts of their choice. This model gives the attacker a great deal of flexibility over the other models and a better chance at recovering a key.

Oracles

An oracle is a generic term that refers to something an attacker is able to query for information. Oracles are commonly used to gain information about plaintext data or encryption keys. An oracle is almost always some aspect of the system being attacked. Thus, an oracle queries contextually relevant information.

The query itself does not have to be executed or received directly, either. As long as you can consistently make the query and get the result through any means, the oracle is valid. This makes oracles very popular, adaptive methods of attack.

An Introduction to Modern Cryptography, 2nd Edition, By Katz and Lindell, contains excellent examples and formal definitions of different types of oracles.

Encryption Oracles

An encryption oracle gives the attacker the ability to encrypt plaintexts of their choice and receive the ciphertext in return. Here the attacker uses a plaintext to query the oracle, which encrypts the plaintext using an unknown key, and exposes the ciphertext. The chosen-plaintext attack is an example of an attack that uses an encryption oracle.

Decryption Oracles

You have probably already figured out that a decryption oracle is the opposite of an encryption oracle. Attackers can query a decryption oracle for a plaintext using a ciphertext. The chosen-ciphertext is an example of an attack that uses a decryption oracle.

Padding Oracles

A padding oracle is a type of decryption oracle. However, instead of querying the oracle for a plaintext value, the oracle is only queried for an indication of whether or not the decryption succeeded. Block cipher modes that rely on padding to make a plaintext message a perfect multiple of the cipher's block length, like CBC Mode, can be susceptible to divulging the entire contents of a message through a padding oracle. This can be an *extremely* serious attack.

Verification Oracles

Verification oracles allow the attacker to attempt to verify some type of data such as a hash or a MAC. The oracle lets the attacker know whether the verification process succeeded or failed, and in some cases, the time it took to fail or at which point in the process the failure occurred. Where a verification oracle can accurately gauge the underlying function's timing, this can open the door to a *timing attack*.

Collision Attacks

Collision attacks exploit the probability of finding a collision in data within a finite space. What we are talking about here are pieces of data like hashes, keys, and numbers that can only be 1 of N possible values. A 128-bit key exists within a key space that has 2^{128} possible values. Therefore, any 128-bit key (or any 128-bit piece of data) is one of those finite values. However, while these spaces (128-bit or 256-bit, for example) can be very large, collisions between data that exist within them happen fairly often. There are two main types of collision attacks: birthday attacks and meet-in-the-middle attacks.

Birthday Attacks

If there are 23 people in a room, the probability exceeds 50% that two people share the same birthday. There are only 365 possible birthdays. What we have here is a collision of values in a finite space. This concept is most often applied generically to hash functions, where an N-bit hash function will generate a collision after approximately $2^{n/2}$ different inputs. However, the birthday attack can be used to determine the likelihood of a collision in any finite space using the same formula—about $\sqrt{N}$, or $2^{n/2}$, where N is the bit size of the space. For example, SHA-256 is a hash function that produces a fixed-length 256-bit output given a variable-length input. A collision occurs where input values render the same output, or hash. The birthday attack tells us that we can expect a collision in SHA-256 within around 2^{128} inputs.

We can also see the birthday attack apply to things like random number generation. Like before, we can expect collisions in around the root of the possibilities. So, take for example a 32-bit number. If we are randomly generating 32-bit numbers using a random number generator (RNG), we can expect the RNG to produce a collision within 2^{16}. Remember that this applies to a collision among *any* numbers that it has generated, not a *specific* number that we are holding. This is very important for things like symmetric key and initialization vector (IV) generation. In these practical scenarios, we can expect the same random key generator to generate the same 128-bit key twice around 2^{64} generations. Therefore, the birthday attack is very important to keep in mind when designing systems and considering their actual strength.

Meet-in-the-Middle (MITM) Attacks

A *meet-in-the-middle* is another type of collision attack that is more flexible than birthday attacks. This is not to be confused with *man-in-the-middle* attacks that serve to insert the attacker between legitimate parties when they communicate.

Let’s start with a scenario where we’re trying to compromise a 64-bit MAC key for which we know the message, such as “Welcome to ABC Bank!” With a MITM we would start our attack by generating 2^{32} different 64-bit keys at random. For each key that we generate, we would compute the MAC on “Welcome to ABC Bank!” and store the key and MAC tag in a database. Our goal now is to capture as many packets as possible and compare the MAC to those that we have computed and stored in the database. If we have a match, we can assume the keys match, and we would then be able to generate a MAC for a message of our choice that would be seen as valid. We can expect a match in approximately 2^{32} packets. This puts our total work factor at 2^{33} to compromise a 64-bit key, which isn’t bad.

Exhaustive Search Attacks

Exhaustive search attacks try all possible values within a target space. Therefore, if data is protected by a 128-bit encryption key, it would take a maximum of 2^{128} attempts to find the correct key. In an exhaustive search attack, the attacker has a greater than 50% chance of finding the target value once half of the key space has been attempted, or 2^{N-1} possibilities where N is the size of the space. So, in a 128-bit space, this would be 2^{127}.

The prospect of inevitably recovering a target value in exhaustive search attacks can lead attackers to consider them viable options where the workload is practical. This can be the case for short value lengths like 56-bit DES keys, which are no longer considered secure. However, while exhaustive search attacks can be attempted for larger value lengths, such as 128-bit or 256-bit, these encompass such a vast number of possibilities that they’re not considered viable routes for an attacker given current computing resources and technologies. Even a device that could check a *billion billion* values (10^{18}, or *1,000,000,000,000,000,000*) per second would still require about 10^{13} years to exhaust a 128-bit target space.

Side-Channel Attacks

Side-channel attacks seek to exploit peripheral aspects of a system or an algorithm that expose useful information to the attacker. Traffic analysis, timing attacks, radiation and frequency emission attacks, and power attacks, to name a few, are all examples of *side-channels* which can be used to gain additional information about a system and its components.

Traffic Analysis

Traffic analysis is a generic term for studying a communication channel to learn which parties are communicating, with whom they are communicating, and the frequency and size of the messages. Traffic analysis can uncover very valuable data; in fact, a lot can be learned from studying the pattern of messages within a communication channel as whole. Regardless of what is being said, sometimes just the fact that two parties are communicating, or even at particular times, is a huge piece of information.

For example, when we see party A transmit a message to party B, who promptly contacts 8 additional parties, and this happens with regular frequency, this may signify a chain of command. Even where parties communicate over encrypted channels, it's still easy to notice some of the communication patterns that we rely on in our day-to-day lives. In *Secrets and Lies*, Bruce Schneier gave an interesting example of what traffic analysis can offer:

> *In the hours preceding the U.S. bombing of Iraq in 1991, pizza deliveries to the Pentagon increased one hundredfold. Anyone paying attention certainly knew something was up.*

Timing Attacks

The amount of time a particular cryptographic operation takes can give the attacker information about the underlying data being processed or the state of the operation. A generic *timing attack* could be executed using any temporal aspect of a system that an attacker can find and exploit.

Verification Timing Attacks

An example that we will use later in the book is a byte array comparison method used for comparing cryptographic hashes or message authentication codes (MACs). If the method fails at the first byte comparison that doesn't match between two values, an attacker could use the method to gain information about the underlying hash based on the time it takes to compare against their input data. However, a method yielding a flat comparison time will take away the attacker's ability to do this.

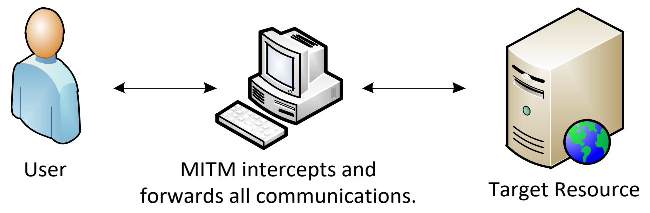
Timing Attacks on Keys

Some timing attacks have been able to recover private asymmetric keys by monitoring how long certain cryptographic operations have taken to execute. Others have been able to recover private asymmetric key information by monitoring the timing associated with the generation of a key pair.

Man-in-the-Middle Attacks

Man-in-the-Middle (MITM) attacks are a generic type of networked attack. In an MITM attack the attacker inserts himself or herself between a legitimate user and a target resource such as a server or another user. The attacker tricks both endpoints into believing they are communicating directly with the other, when really they are both communicating with the attacker, who is merely relaying their communications. Therefore, all data the MITM intercepts will be seen in plaintext. Figure 14 shows the MITM relationship between two endpoints.

Figure 14: MITM intercepts communications



MITM attacks pose a serious threat to transport security. A successful MITM will circumvent the security provided by protocols that only protect data in-transit between two endpoints where trust has not been established. However, the MITM will not know a pre-shared key established before the session, which can provide a safeguard in addition to strong trust provided by trusted certificate authority (CA) over SSL/TLS.

Replay Attacks

A replay attack is a generalized networking attack that retransmits an old message to a receiver in an attempt to trick the receiver into believing the message is valid (or was just transmitted in real time). Replay attacks have been used in all types of scenarios to dupe a system into accepting and processing an old message. Money transactions that transmit a balance or a current price are common targets. So are systems that use a standard messages to change passwords or elevate privileges or access.

There are a few simple, fundamental ways to prevent replay attacks. One is to establish sequence numbers or message numbers so that a replayed message will be detected. Another common method of prevention is to use a one-time token or nonce that is transmitted with a packet. This is another reason to use a trusted, vetted protocol when available that already implements sequence numbers to protect against replay attacks.

The Human Factor: Social Engineering Attacks

Social engineering involves exploiting weaknesses in people. These types of attacks are often generalized because of their extremely broad scope, but can include cons, threats, bribery, blackmail, espionage, and physical violence. A simple scenario would be an attacker simply tricking a customer service representative into providing a user’s password by convincing the representative that he is the user. Answers to security questions are often posted in social media: pet names, school and dating history, personal preferences, and other unknowingly harmful info.

Many of the attacks we see in the media are a result of social engineering. Where information like passwords are needed to compromise extremely valuable assets, attackers may be more inclined to resort to more serious social engineering attacks to gain access, like bribery or blackmail. This can be more common in the case of national or corporate espionage.

Denial-of-Service (DOS) and Distributed Denial of Service (DDOS)

Denial-of-service (DOS) attacks intend to disrupt the availability of a system. Another variety, distributed-denial-of-service (DDOS) attacks, use multiple machines to execute the DOS. Most of these attacks target and overwhelm a system to block or disrupt legitimate users. However, other methods that disrupt communications—cutting power, cutting wiring, attacking another entity necessary for the communications to take place—are also practical DOS attacks. Despite being considered a more general security issue, DOS is a very real threat to cryptographic systems where users' inability to successfully protect or process information could lead to a breakdown in secure communication or access to secure data.

Rootkits and Viruses

This is a very broad category that can encompass many of the other attacks that we've covered. Rootkits and viruses have been used to compromise and circumvent cryptographic protections with huge success (API hooking is also an effective attack). The level of access that these attacks have makes them extremely potent; they have the ability to obtain encryption keys and passwords without the need to employ a traditional cryptographic attack (CCA, CPA, oracles, brute force).

Physical Attacks

Physical attacks are those that are executed physically against a system or its components. These attacks require physical access to a system and are generally the most dangerous. This is often the caveat in security systems, and is one of the primary reasons why data centers have such high security. Always employ physical safeguards for your systems such as vaults, safes, locked cabinets, mantraps, surveillance, and security guards.

Chapter Summary

Cryptography has been used throughout history to secure sensitive communications and data. Cryptography has grown tremendously in the last century, most notably in the 1970's when asymmetric encryption solved many of the problems posed by symmetric encryption. Today, cryptographic primitives are used to achieve particular aspects of security, often within a larger system. Understanding the uses for different types of cryptographic algorithms is important when designing secure applications and solving security-related problems. Cryptographic key management is one of the most difficult aspects in designing cryptographic systems and should be a major design consideration.

Depending on the application and context, implementations of standardized cryptographic protocols, such as SSL/TLS, can and should be used when available. Not only is this almost always the more secure choice, it's usually the more efficient one, too.

There are many well-known attack models used against cryptographic systems. Some of these focus on compromising a particular algorithm or type of cryptographic primitive; others are more generic. It's important for developers to understand the basics of these attacks and their targets. Despite the security that many of the popular cryptographic algorithms offer, poor implementation can open the door to a variety of attacks.

Chapter Questions and Exercises

1. Explain the applications of symmetric and asymmetric encryption, cryptographic hash algorithms, message authentication codes (MACs), digital signatures, and random number generators (RNGs).
2. What are the major benefits of public key cryptography?
3. What are the differences between block ciphers and stream ciphers?
4. How are MACs and digital signatures different? How are they similar?
5. What is cryptographic key management?
6. Explain the various types of attacks and their purpose.
7. Explain the relationship between collision attacks and cryptographic hash algorithms.

2 .NET Cryptography

Chapter Objectives

1. Learn about the various cryptography namespaces in .NET and what they contain.
2. Understand the class hierarchy used in .NET for cryptographic algorithms.
3. Learn how the **CryptoConfig** class is used to configure mappings.
4. Recognize common auxiliary libraries not included in .NET and where the CryptoAPI fits into the .NET cryptography model.
5. Know the cryptographic exceptions used in .NET.
6. Appreciate the challenges presented by memory management and text encoding.

What's New in .NET 4.6?

Most of the cryptographic functionality from .NET 4.5 remains the same. There are a few notable exceptions:

- **ECDsaCertificateExtensions** (System.Core.dll)**:** Provides easier access to certificate information and keys.
- **RSACertificateExtensions** (System.Core.dll)**:** Provides easier access to certificate information and keys.
- **RSACng** (System.Core.dll): A Cryptography Next Generation (CNG) implementation of the RSA algorithm.
- **RSAEncryptionPadding:** Specifies the padding mode (Oaep (SHA1-SHA512) or Pkcs1) and parameters to use with RSA encryption or decryption operations.
- **RSAEncryptionPaddingMode:** An enum that specifies the encryption padding mode (Oaep (SHA1-SHA512) or Pkcs1).
- **RSASignaturePadding:** Specifies the padding mode (Pkcs1 or Pss) and parameters to use with RSA signature creation or verification operations.
- **RSASignaturePaddingMode:** An enum that specifies the signature padding mode (Pkcs1 or Pss).

mscorlib.dll and System.Core.dll

Cryptographic functionality and resources in .NET are organized into specific namespaces and dlls (or made available through COM). **mscorlib.dll** and **System.Core.dll** contain much of the core functionality of .NET and most, if not all, of the cryptographic objects needed to perform essential cryptographic programming in .NET applications. **mscorlib.dll** and **System.Core.dll** contain objects in the **System.Security.Cryptography** and **System.Security.Cryptography.X509Certificates** namespaces that we will use throughout this book.

System.Security.dll

The **System.Security.dll** offers additional functionality atop that of the **mscorlib.dll**. Notable additions are the **System.Security.Cryptography.Xml** and **System.Security.Cryptography.Pkcs** namespaces, and the **DpapiDataProtector**, **ProtectedMemory** and **ProtectedData** objects within the **System.Security.Cryptography** namespace.

System.Security.Cryptography

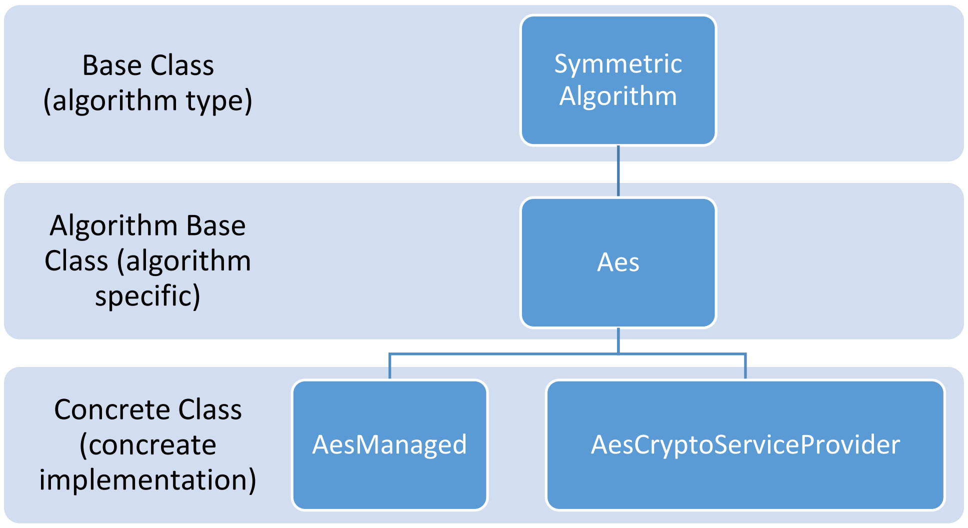
The **System.Security.Cryptography** namespace is the primary namespace for accessing cryptographic resources and objects in .NET. This namespace contains implementations of popular algorithms for cryptographic hashing and keyed hashing, random number generation, encryption, and digital signing. Examples throughout the book will assume that you have it referenced in your code file:

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

Class Hierarchy

Most of the classes that we will be working with have an inheritance pattern similar to what you see in other parts of .NET. For example, all symmetric encryption algorithms in .NET implement the **SymmetricAlgorithm** abstract class. Additionally, every algorithm has its own abstract class, from which concrete implementations inherit. Let's use AES as an example. AES, a symmetric encryption algorithm, can be used through the **AesManaged** and **AesCryptoServiceProvider** concrete classes. Both of these classes inherit from the algorithm-specific abstract class, **Aes**. **Aes** is a symmetric algorithm, which means it implements the **SymmetricAlgorithm** abstract class. Thus, **AesManaged** and **AesCryptoServiceProvider** implement **SymmetricAlgorithm** and can be used in a polymorphic fashion. Figure 15 shows this hierarchy.

Figure 15: Class Hierarchy for Symmetric Algorithms in .NET


Base Class (algorithm type)
Symmetric Algorithm
Algorithm Base Class (algorithm specific)
Aes
Concrete Class (concreate implementation)
AesManaged
AesCryptoServiceProvider

Common Classes and Objects

Table 4: Algorithm base classes outlines types of cryptographic algorithms and corresponding base classes in the **System.Security.Cryptography** namespace.

Table 4: Algorithm base classes

Type of Algorithm	Algorithm Specific Base Class
Cryptographic Hash Algorithms	MD5, RIPEDMD160, SHA1, SHA256, SHA384, SHA512
Symmetric Encryption Algorithms	Aes, Rijndael, TripleDES, RC2, DES
Message Authentication Codes	HMAC
Asymmetric Encryption and Key Exchange Algorithms	RSA, ECDiffieHellman
Digital Signature Algorithms	RSA, DSA, ECDsa

CryptoConfig

Cryptographic algorithm mappings for .NET are contained in the *maching.config* file in the *C:\[Windows]\Microsoft.NET\Framework\[Version]\Config* directory. This can be adjusted manually through XML or using the **CryptoConfig** class in the **System.Security.Cryptography** namespace. We will show you how to use the latter. However, there should be plenty of online resources available for how to manually edit and add to the *machine.config* file.

The **CryptoConfig** class provides access to mapping between a .NET cryptographic algorithm and its name or OID number within the *machine.config* file. Table 5 describes members of the **CryptoConfig** class.

Table 5: Members of the CryptoConfig class

Member	Description
AddAlgorithm	Adds a mapping between a type and a string name within the current application domain.
AddOID	Adds a mapping between an OID and a string name within the current application domain.
CreateFromName	Creates a new instance of a cryptographic object given a string name that is mapped to a type.
EncodeOID	Encodes an OID string as a byte array.
AllowOnlyFipsAlgorithms	A read-only bool indicating whether the runtime should implement a policy enforcing Federal Information Processing Standard (FIPS) algorithms.

Creating an Instance from a Name

Creating an instance from a string name is performed with the **CreateFromName** method, which invokes the mapping between a string name and the algorithm type. Because this method returns an **object**, you will need to cast it. Below, we create an instance of **AesManaged** using the default name for this algorithm, which is "AesManaged":

```
AesManaged myAlg = (AesManaged)CryptoConfig.CreateFromName("AesManaged");
```

Alternatively, we could cast to its base class, **SymmetricAlgorithm**:

```
SymmetricAlgorithm alg = (SymmetricAlgorithm)CryptoConfig.CreateFromName("AesManaged");
```

This is also the mapping that many of the cryptographic base classes rely on when creating an algorithm instance from a name. For example, we could create an instance of **AesManaged** using the **CreateFromName** method, or alternatively, the **SymmetricAlgorithm.Create(string)** or **Aes.Create(string)** methods.

Changing Algorithm Name Mappings

The **AddAlgorithm** method allows developers to add a new mapping for a cryptographic algorithm or override an existing one. These mappings are used within an application domain when creating an algorithm instance as discussed last section.

The syntax for **AddAlgorithm** is straightforward: the first parameter takes the type of the object being mapped; the second parameter accepts the string name to use in the mapping. Below, we create a mapping between the **AesManaged** type and the name “myAlg”:

```
CryptoConfig.AddAlgorithm(typeof(AesManaged), "myAlg");
```

Now, we could create an instance of **AesManaged** using the **CreateFromName** method with the name “myAlg”:

```
AesManaged aes = (AesManaged)CryptoConfig.CreateFromName("myAlg");
```

Working with OIDs

Object Identifiers (OIDs) are numbers used to uniquely identify cryptographic algorithms. The **MapNameToOID** method provides a mapping between an algorithm name and its OID. However, you will have to use the algorithm name itself, **RSA**, instead of the concrete class name in .NET, which is **RSACryptoServiceProvider**. Below we get the OID for RSA, which is *1.2.840.113549.1.1.1*:

```
string oid = CryptoConfig.MapNameToOID("RSA");
```

Another method, **EncodeOID**, is used to encode an OID into a byte array format. Here, we’ll use the OID variable from above:

```
byte[] encodedOid = CryptoConfig.EncodeOID(oid);
```

CryptoStreams

Streams allow data to be accessed and handled serially, rather than all at once (like using an array which must be contained in memory). Streams typically maintain a rather sleek profile because large pieces of data, such as files, can be processed in smaller pieces rather than being pulled into memory all at once.

There are two types of streams in .NET stream architecture: *backing store streams* and *decorator streams*. *Backing store streams* provide the actual endpoints necessary to interact with the resources themselves such as **FileStream**, **MemoryStream**, or **NetworkStream**. *Decorator streams* are used in conjunction with another stream object to transform the stream data in some manner. They can be chained to other decorator stream objects for increased functionality. The **CryptoStream** object in .NET is a type of decorator stream. **CryptoStream** objects are used to perform cryptographic operations on *stream* data. **CryptoStream** objects should be used on backing store streams where it would be inefficient or difficult to handle the resource in memory as a byte array.

Encryption Options in.NET

There are many ways to perform encryption and hashing in .NET. Table 6 contains the different options in .NET and their relative strength, speed, and ease of use:

Table 6: Encryption Options in .NET

Option	Ease of Implementation	# of Keys to Manage	Speed	Strength
File.Encrypt	Easy	0	Fast	Hinges on access to account/user password
Windows Data Protection (DPAPI)	Easy	0	Fast	Hinges on access to account/user password
Symmetric Encryption	Medium	1	Fast	Strong
Public Key Encryption	Medium	2	Slow	Strong
Public Key Hybrid Encryption (public key and symmetric)	Medium	3	Medium/Fast	Strong

System.Security.Cryptography.Xml

The **System.Security.Cryptography.Xml** namespace is located in **System.Security.dll** and provides functionality for working with standardized XML cryptography in .NET. This namespace uses .NET implementations of cryptographic algorithms in the **System.Security.Cryptography** namespace where cryptographic operations are required.

System.Security.Cryptography. X509Certificates

The **System.Security.Cryptography.X509** namespace gives developers the means to retrieve basic information from an X509 certificate and import and export certificate material.

The Underlying Crypto API

The Crypto API (CAPI) is a Windows API. CAPI provides cryptographic functionality and basic key storage and management. Parts of .NET use CAPI to obtain implementations of standard algorithms as well as persist asymmetric key pairs to the CAPI Key Store.

CAPICOM

CAPICOM is a COM wrapper for the Crypto API. Its capabilities include encryption, signing, and basic key storage, with extensive uses for certificates.

Web Services Enhancements (WSE)

In terms of cryptography, the Web Services Enhancements (WSE) for .NET afforded better options for working with certificates and accessing system certificate stores than the **System.Security.Cryptography.X509** namespace in earlier versions of .NET. Newer versions have significantly expanded on the functionality of the **System.Security.Cryptography.X509** namespace.

WSE is available for download (at the time of writing) at:

http://msdn/webservices/building/wse

Enterprise Library Cryptography Application Block

Rewriting the same boilerplate code to perform basic tasks like symmetric encryption and hashing is inefficient. The Enterprise Library Cryptography Application Block (ELCAB) from Microsoft is an attempt to simplify the implementation of cryptography in applications by abstracting away boilerplate code through a simple and extensible interface. ELCAB can be used to perform symmetric encryption, hashing, and customized key management. It's available as a *Nuget* package, but we will not cover it in this book.

High Encryption Pack

For years there were export restrictions in place for encryption. This applied specifically to key lengths for encryption algorithms. Manufacturers had to reduce key sizes that algorithms used for the operating systems and products they shipped. As a result, domestic consumers who wanted to benefit from stronger encryption had to download higher encryption support. For Windows 2000 this meant users needed the High Encryption Pack, which was available with Service Pack 2 (SP2), or available separately for download. However, all OS versions since Windows XP and Windows .NET Server have included support for larger keys.

Cryptographic Exceptions

.NET has two types of cryptographic exceptions that you will probably be seeing in some of your own examples and apps as you work through the book material.

- **CryptographicException** is thrown when an *error* occurs during a cryptographic operation. These often result from invalid inputs for algorithms.
- **CryptographicUnexpectedOperationException** is thrown when an unexpected operation occurs during a cryptographic operation.

Text Encoding

Many algorithms and objects in .NET only accept data in a byte-array or stream format. Many of the communications that we are expected to protect, however, are in a string (text) format. This requisite change in format opens the door to text encoding issues. This can be especially challenging between environments that perform encoding differently.

Follow the following steps to avoid encoding issues when encrypting and processing string data:

1. Convert any string data you need to encrypt will need to byte array format before encryption. UTF8 will usually work fine and is a default for .NET.
2. Perform the encryption, which will return a ciphertext byte array.
3. If you need to persist or transmit the ciphertext in text format, use Base64 encoding.
4. Before decryption, if the ciphertext is Base64-encoded, convert it back to a byte array. Remember, Base64 is a stable type of encoding, but will occupy 1/3 more space.
5. Perform decryption, which will return the plaintext byte array.
6. Use the same type of text encoding from step 1 to return the byte array to string format.

Memory Management

Sensitive information resides in memory during cryptographic operations. In the simplest scenarios this is plaintext getting passed through a hash algorithm. But the most common scenarios are those that expose cryptographic keys and passwords in addition to sensitive plaintext. In a secure environment, sensitive data in memory should be given a narrow scope and wiped when this scope expires, which usually involves clearing a byte array. This reduces the risk of sensitive memory contents being moved to persistent storage (disk) in a swap file, or being pulled or snapshotted, and ultimately obtained by an attacker.

.NET provides a managed environment. The .NET runtime's Garbage Collector frees allocated memory when deemed appropriate. Consequently, developers do not have a mechanism to deterministically manage memory like they would in languages like C or C++. But even after memory has been freed by garbage collection, the memory location has not been cleared or zeroed. In .NET, resources handling sensitive data should have their sensitive memory cleared prior to turning it over to the garbage collector. Many cryptographic objects have a **Clear** or **Dispose** method (if they implement **IDisposable**) that will clear sensitive memory used by keys or plaintexts.

Another hurdle for developers in .NET is string immutability. What this means is that a string that exists in memory cannot be changed or cleared by the developer. Any change made to an existing string value simply creates and references a new version in memory, leaving the old one for garbage collection. So, if you have secret data held in a string:

```
string secret = “This is a secret message”;
```

This data will remain in memory regardless of what you do to the variable's value. If you set the string to null or an empty string, as shown below, it will not actually destroy or alter the sensitive data in memory. It will only create a new string in memory each time it's changed and leave the old one to be picked up by garbage collection.

```
secret=null;
```

```
secret="";
```

It should be clear that strings are not a secure data type and their role should be minimized as much as possible in secure application programming.

Sensitive data and keys should be handled in byte array format whenever possible so that the memory can be protected prior to use and destroyed (cleared, disposed, or zeroed) after. However, this isn't always possible. Where user data from a GUI has to be captured in string format you won't be able to secure it in memory. For best security you'll have to keep track of the sensitive memory, minimize its exposure, and do your best to wipe it when you're done.

Chapter Summary

- .NET offers a stable and intuitive cryptography model. The primary cryptographic namespace in .NET is **System.Security.Cryptography**.
- Much of the .NET cryptographic functionality is built upon the underlying Crypto API.
- CAPICOM, WSE, and the Enterprise Library Cryptography Application Block (ELCAB) can all be used to achieve greater cryptographic functionality in .NET. But remember that some of these options are unmanaged or no longer supported.
- .NET uses two types of cryptographic exceptions: **CryptographicException** and **CryptographicUnexpectedOperationException**.
- Secure memory management in .NET can be difficult because of the managed memory model and garbage collection. String data in .NET is immutable and cannot be cleared or destroyed in any deterministic fashion.

Chapter Questions and Exercises

1. What is the primary namespace used for cryptography in .NET?
2. Explain the class hierarchy for cryptographic algorithms in .NET.
3. What is the **CryptoConfig** class used for?
4. What are some of the security issues presented by string data in .NET? Why is byte-array data considered safer than string data?

3 Randomness and Random Number Generators

Random Number Generation: *A process used to generate an unpredictable series of numbers. Also, referred to as a Random bit generator (RBG).*

Chapter Objectives

1. Understand why the **Random** class is not cryptographically secure.
2. Learn how to create cryptographically secure random data using the **RNGCryptoServiceProvider** class.
3. Understand how random number generators can be susceptible to collisions and the birthday paradox.
4. Recognize the applications for randomness in cryptography and common uses for random numbers/data.

The first cryptographic primitive we will cover is the Random Number Generator (RNG). Randomness plays a critical role in the successful implementation of cryptographic solutions. The biggest demand for random material comes from symmetric (private) keys, block cipher initialization vectors (IVs), nonce (number only used once) values, and salts (discussed in chapter 6). Finding a source that consistently produces unpredictable and random material, however, can be harder in practice than in theory.

Random. Pseudo-random. Cryptographically random. The terminology used by programmers to describe randomness can be confusing and misleading. Random material is needed in cryptographic solutions to provide security and unpredictability. Most developers have been introduced to their language's or framework's *Random* class as a source for *random* material. The problem that most developers are unaware of is that most "Random" classes do not actually produce random data; they produce data that appears random, but is actually *predictable*. These classes are not secure for use in production cryptosystems. For cryptographic systems that rely on this type of randomness to generate keys, IVs, or nonce values, this could be a catastrophic flaw, and is one that has led to major exploits.

The Problem with the Random Class

The **Random** class is often used for general-purpose programming that requires random data. C++ developers are probably familiar with the **rand** class, while C# uses a class called **Random**. Both of these are insecure for cryptographic purposes because the material generated is predictable to the extent that once a sequence of this data is known, an attacker can actually determine the next sequence to be generated. Although this is typically well known among security professionals, these classes are constantly implemented in secure solutions to generate important pieces of randomness. Regular (insecure) random classes should never be relied upon for this purpose.

The C# code below is a good example of how NOT to use the **Random** class in cryptosystems. This code was actually used to generate keys as well as IVs for an AES encryption scheme used to secure personally identifiable information (PII) in a web application.

```
byte[] randomData = new byte[16];

Random rand = new Random();

rand.NextBytes(randomData);
```

Cryptographically secure random number generators—as a general rule of thumb—can usually be found in crypto namespaces and libraries, but developers should always research these classes and their sources of random material before implementation in production environments. .NET developers can turn to the **RNGCryptoServiceProvider** class in the **System.Security.Cryptography** namespace for secure random material. An explanation of this class's functionality will be covered next.

Generating Cryptographically Secure Random Material in .NET

In .NET, the **RNGCryptoServiceProvider** class has a simple interface that can be used to generate cryptographically secure random material. After creating a new instance of the class, the **GetBytes** method will fill a byte array with a cryptographically secure random sequence of bytes. The **GetNonZeroBytes** method has a nearly identical usage but will disallow zero-value bytes in the sequence.

The example below shows how to use the **GetBytes** method to fill a byte array with a sequence of random material:

```
byte[] randomData = new byte[16];

RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

rng.GetBytes(randomData);
```

Example: Testing RNG Collisions Using the Birthday Paradox

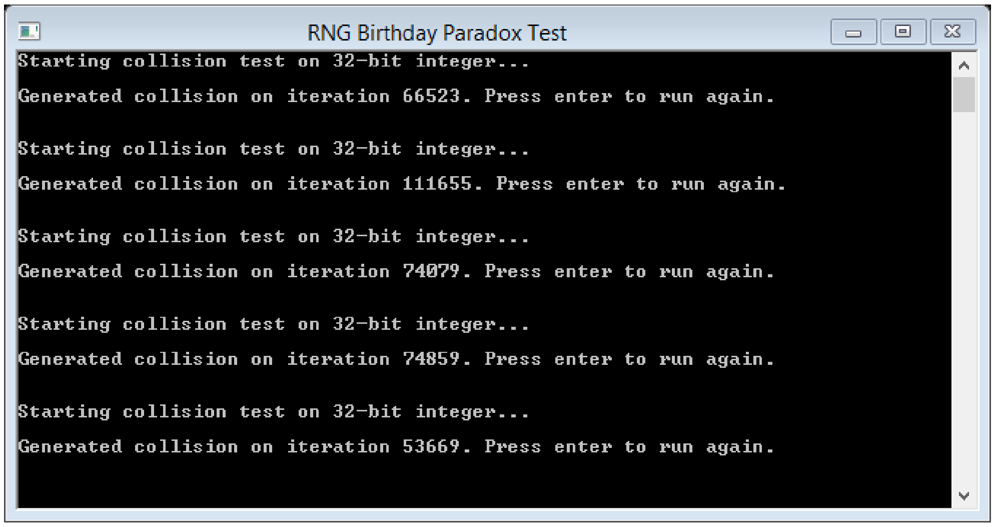
RNGCryptoServiceProvider generates cryptographically strong random data. This does not mean that it won't generate the same data again, even quickly. For example, take a 32-bit integer. This has 2^{32} (4,294,967,296) possible values. The birthday paradox involves finding collisions in values that live within a finite space. The odds of finding a collision in data exceed 50% when a random sample size reaches approximately the root of the possibilities. Therefore, a collision can be expected in randomly generated 32-bit integers in approximately 2^{16} (65,535) attempts.

A quick n' dirty console app can test this theory for us:

```
static void Main(string[] args)
{
    RNGCryptoServiceProvider generator = new RNGCryptoServiceProvider();

    Console.WriteLine("Starting collision test on 32-bit integer...\n");

    List<int> attempts = new List<int>();

    for (int i = 0; ; i++)
    {
        byte[] attempt = new byte[4];

        generator.GetBytes(attempt);

        int attemptNum = (int)BitConverter.ToInt32(attempt, 0);

        if (attempts.Contains(attemptNum))
        {
            Console.WriteLine(String.Format("Generated collision on iteration {0}. Press 
enter to run again.\n \n", i.ToString()));
            break;
        }

        attempts.Add(attemptNum);
    }

    Console.ReadKey();

    Main(null);
}
```

Figure 16 shows collision results when the app runs (this may take seconds/minutes depending on your machine).

Figure 16: Running the RNG Birthday Paradox Test


RNG Birthday Paradox Test

```
Starting collision test on 32-bit integer...
Generated collision on iteration 66523. Press enter to run again.

Starting collision test on 32-bit integer...
Generated collision on iteration 111655. Press enter to run again.

Starting collision test on 32-bit integer...
Generated collision on iteration 74079. Press enter to run again.

Starting collision test on 32-bit integer...
Generated collision on iteration 74859. Press enter to run again.

Starting collision test on 32-bit integer...
Generated collision on iteration 53669. Press enter to run again.
```

The program actually generates collisions pretty close to our target. Remember, we have a 50% chance of generating a collision around 65,535 attempts. The results might look a little disparate, but when looking at the big picture all of the results have landed between 2^{15} and 2^{17} attempts, which is fairly accurate.

Most modern production systems will be using symmetric keys between 128 and 256 bits. Given the birthday paradox, if keys are being randomly generated, you can expect a duplicate key to be generated around 2^{128} attempts for a 256-bit key and 2^{64} attempts for a 128-bit key. We will be talking about the birthday paradox throughout the book and how it relates to different issues.

Example: Assessing Maximum Password Entropy

Sometimes it's important to know the maximum possible bit strength of a plaintext password. Where we have passwords being used to protect data or provide access control to resources, the maximum level of protection is the strength of the password. The maximum bit security level that a password can carry is determined by the available pool size of its characters and the password length. For example, the pool size for a numeric pin, regardless of length, is 10 (0-9). Similarly, case insensitive alpha (A-Z or a-z) has a pool size of 26. Table 7 shows the necessary length to derive 128-bit and 256-bit keys given pool size.

Table 7: Requisite password lengths to derive 128-bit or 256-bit keys

	Pool Size (# of available characters)	Requisite password length to derive a 128-bit key	Requisite Password length to derive a 256-bit key
Numeric Pin	10 (0-9)	40	77
Case-insensitive alpha	26 (A-Z) or (a-z)	28	55

Case-sensitive alpha	52 (Aa-Zz)	23	45
Case-sensitive alpha plus numeric	62 (Aa-Zz and 0-9)	22	43
Case-sensitive alpha plus numeric and punctuation	93 (Aa-Zz, 0-9, and punctuation)	20	40

To calculate the max bit strength for any password, we can use the following method. It accepts pool size and length as arguments and returns an integer representing the strength:

```
static int GetMaxEntropyBits(int poolSize, int length)
{
    if (poolSize == 0) return 0;

    return Convert.ToInt16(length * Math.Log(Convert.ToDouble(poolSize)) /
Math.Log(Convert.ToDouble(2)));
}
```

For a 12 character, case-sensitive, numeric password with punctuation, we could call it like this (which returns 78):

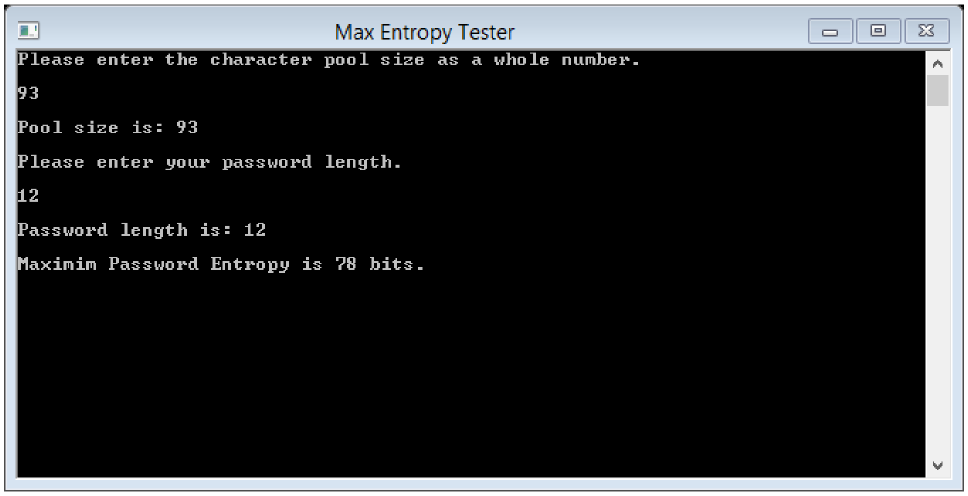
```
int maxStrength = GetMaxEntropyBits(93, 12);
```

Next, we'll implement this method in a simple console application:

```
static void Main(string[] args)
{
    Console.Title = "Max Entropy Tester";

    Console.WriteLine("Please enter the character pool size as a whole number.\n");

    int poolSize = Convert.ToUInt16(Console.ReadLine());

    Console.WriteLine("\nPool size is: " + poolSize.ToString()+"\n");

    Console.WriteLine("Please enter your password length. \n");

    int passLength = Convert.ToUInt16(Console.ReadLine());

    Console.WriteLine("\nPassword length is: " + passLength + "\n");

    Console.WriteLine("Maximim Password Entropy is " + GetMaxEntropyBits(poolSize,
passLength).ToString()+" bits.");

    Console.ReadKey();

    Main(null);
}
```

As you can see in Figure 17, running the app with a pool size of 93 and a password length of 12 outputs a maximum password entropy of 78 bits.

Figure 17: Running the Maximum Entropy Tester app



```
Please enter the character pool size as a whole number.
93
Pool size is: 93
Please enter your password length.
12
Password length is: 12
Maximim Password Entropy is 78 bits.
```

Recommendations

Always track down where and how your random data is being generated. The most secure option currently in .NET is the **RNGCryptoServiceProvider** class. It's good policy to determine if your password is capable of containing the requisite entropy for your security strength. For instance, if a password's strength is limited to 90 bits due to pool size and length, this will essentially serve as a security bottleneck for a system that is trying to deliver a 128-bit strength.

Chapter Summary

- Entropy is the measure of randomness.
- Maximum password strength is determined based on character pool size and password length.
- Pseudo-random number generation in the **Random** class is not secure for cryptographic applications because of its predictability. **Random** should not be used to generate any material used to secure data, including keys, IVs, nonce values, or salts (these will be addressed later in the book).
- The **RNGCryptoServiceProvider** class is a secure random number generator and should be used to generate random material for things such as cryptographic keys, IVs, nonce values, and salts.

Chapter Questions and Exercises

1. Why is the **Random** class not suitable for cryptographic purposes? Which class should be used instead?
2. Why are random number generators susceptible to the birthday paradox when generating fixed-sized data? What would happen if they weren't? Would they be more or less secure for cryptographic purposes?
3. Write a program that seeks to determine the difference between random data from a human and that of the **RNGCryptoServiceProvider** class using the frequency stability property from page 9.

Scenarios

1. You need to generate random material for consumption by cryptographic objects. Some of the material will be used for secret keys, and some will be used for initialization vectors (IVs). One of your team members on the project has already written a lot of the code using the **Random** class in knowing disregard for its lower security. His primary reasoning is that the keys are secret anyway and the attackers would still have to figure out a way to guess them. Is his point valid? What should you do?

4 Cryptographic Hash Algorithms

Cryptographic Hash Algorithm: *A function that maps a bit string of arbitrary length to a fixed-length bit string. Approved hash functions satisfy the following properties: 1. (One-way) It is computationally infeasible to find any input that maps to any pre-specified output, and 2. (Collision resistant) It is computationally infeasible to find any two distinct inputs that map to the same output.*

Chapter Objectives

1. Learn the meaning of *collision resistance*, *preimage resistance*, and *second preimage resistance*.
2. Understand why hash algorithm security is assessed in terms of collision resistance.
3. Know the hash algorithms in .NET and which are able to provide at least 128 bits of collisions resistance.
4. Learn the uses of the **HashAlgorithm** base class and how to compute and verify a hash.
5. Understand salting and what it protects against.

A hash algorithm is a function that takes a variable-length input and renders a fixed-length nonreversible output (also called a fingerprint, thumbprint, hash, or digest). The primary characteristic of this functionality is one-way operation; the output cannot be reversed to reveal the input. Not all hash algorithms, however, are considered cryptographically secure. .NET's **GetHashCode** method that is inherited by default by all objects, for example, is not secure for cryptographic purposes.

According to the National Institute for Standards and Technology (NIST), a cryptographic hash function should have the following three properties:

- **Collision resistance:** It is computationally infeasible to find two different inputs to the cryptographic hash function that have the same hash value. That is, if ***hash*** is a cryptographic hash function, it is computationally infeasible to find two different inputs x and x' for which ***hash***(x) = *hash* (x'). Collision resistance is measured by the amount of work that would be needed to find a collision for a cryptographic hash function with high probability. If the amount of work is 2^N, then the collision resistance is N bits. The estimated strength for collision resistance provided by a hash function is half the length of the hash value produced by a given cryptographic hash function, i.e., the estimated security strength for collision resistance is $L/2$ bits. For example, SHA-256 produces a (full-length) hash value of 256 bits; SHA-256 provides an estimated collision resistance of 128 bits.
- **Preimage resistance:** Given a randomly chosen hash value, *hash_value*, it is computationally infeasible to find an x so that *hash(x)* = *hash_value*. This property is also called the one-way property. Preimage resistance is measured by the amount of work that would be needed to find a preimage for a cryptographic hash function with high probability. If the amount of work is 2^N, then the preimage resistance is N bits. The estimated strength for preimage resistance provided by a hash-function is the length of the hash value produced by a given cryptographic hash function, i.e., the estimated security strength for preimage resistance is L bits. For example, SHA-256 produces a (full-length) hash value of 256 bits; SHA-256 provides an estimated preimage resistance of 256 bits.
- **Second preimage resistance:** It is computationally infeasible to find a second input that has the same hash value as any other specified input. That is, given an input x, it is computationally infeasible to find a second input x' that is different from x, such that *hash*(x) = *hash* (x'). Second preimage resistance is measured by the amount of work that would be needed to find a second preimage for a cryptographic hash function with high probability.

To clarify, the difference between preimage resistance and second preimage resistance is that the hash value in preimage resistance is randomly selected, its input is unknown, and you are trying to hash inputs to find a computed hash value that matches the randomly selected one. With second preimage resistance you have a target input (and its hash) and are trying to find another input that produces a hash value matching that of your target.

Similarly, the difference between collision resistance and second preimage resistance is that the *target input* and its corresponding hash are already known under the second preimage scenario; other inputs are introduced to see if their hash matches the target hash. Collision resistance is concerned with *any two inputs* producing the same hash value; it does not operate under the pretenses of having a “target input” or “target hash.”

Applications of Cryptographic Hash Algorithms

Hash algorithms play a pivotal role in technologies that we depend on in our day-to-day activities. Passwords need to be hashed before they are stored. Files need to be hashed so their integrity can be checked. Data that is sent between parties over a network needs to be hashed for assurance that it has arrived intact. Keys have been added to the hashing processes to turn the regular message digest into a Message Authentication Code (MAC) to integrate tamper resistance into message security (discussed later). All of these contexts have different needs and therefore different requirements for their hash algorithms.

Researching current best practices is important in cryptography. Before you use a hash algorithm for a particular task, it’s important that you determine whether the algorithm is suited for the context in which it will be used. You need to assess the purpose and needs of the solution. You will probably need to consider the security level. If you are trying to achieve a 128-bit security level, MD5 won't work despite its 128-bit length. SHA1, a 160-bit algorithm, won't even provide a 128 bits of security in terms of collision resistance. We would need SHA-256—minimum. SHA-384 would be a more conservative estimate, considering that we can only get about $2^{n/2}$ bits of collision resistance out of an algorithm, where *n* is the hash length. Other factors like performance, solution lifespan, environment, and compatibility will also be concerns that require consideration for how your algorithm is suited for your context.

Cryptographic Hash Algorithms in .NET

The **System.Security.Cryptography** namespace contains popular hash algorithms. Each of these algorithms must implement the **HashAlgorithm** abstract class.

MD5

MD5 is a 128-bit hash algorithm developed by Ron Rivest. It offers good speed but is not secure for use in modern cryptographic systems due to its short length and known distinguishing attacks. MD5 is still used for compliance with legacy applications but should never be considered a trusted hash algorithm. Table 8 contains the **MD5** subclassses.

Table 8: MD5 Subclasses

Abstract Class	Sub Classes
MD5	MD5Cng, MD5CryptoServiceProvider

RIPEDMD160

The RIPEDMD160 is a 160-bit hash algorithm developed in Belgium. The RIPEDMD series are strengthened to resist collisions but the shorter length of RIPEDMD160 makes its candidacy for use in secure solutions—especially those in the future—borderline at best. Table 9 contains the **RIPEDMD160** subclasses.

Table 9: RIPEDMD160 Subclasses

Abstract Class	Sub Classes
RIPEDMD160	RIPEDMD160Cng, RIPEDMD160CryptoServiceProvider

SHA1

The Secure Hash Algorithm was designed by the NSA and based on MD4. SHA1 is a strengthened version of the original SHA that suffered from weaknesses. Collisions in SHA1 can theoretically be found in less than 2^{80} attempts and it should not be relied on as a secure hash algorithm for modern systems. Table 10 contains SHA1 subclasses.

Table 10: SHA1 Subclasses

Abstract Class	Sub Classes
SHA1	SHA1Cng, SHA1CryptoServiceProvider, SHA1Managed

SHA-2 Series

The SHA-2 series in .NET include SHA-256, SHA-384, and SHA-512. These are the most secure hash functions offered by .NET and are acceptable for use in modern secure systems. Their longer length makes them more collision-resistant and capable of providing a 128-bit security level under generic attack models. Table 11 lists the collision resistance, preimage resistance, and second preimage resistance strengths (in bits) of the SHA 2 family.

Table 11: SHA-2 Series Strength

	SHA-224	SHA-256	SHA-384	SHA-512
Collison Resistance Strength in bits	112	128	192	256
Preimage Resistance Strength in bits	224	256	384	512
Second Preimage Resistance Strength in bits	201-224	201-256	384	394-512

Another interesting feature of the SHA-2 series algorithms are their bit sizes: they are all twice as large as the Advanced Encryption Standard (AES) key sizes of 128, 192, and 256 bits. This is really great from a practical point of view where we can only expect a collision resistance strength of about half the bit size. SHA2 makes selecting a hash that compliments the bit-security level of a system pretty easy. Just pick a hash that is twice your key size. So, use SHA256 with AES128, or SHA512 with AES256. While not implemented in the **System.Security.Cryptography** namespace, SHA224 is also part of the SHA series and complements the 112-bit key size for 3DES/TDEA. Table 12 contains the subclasses for each of the SHA-2 algorithms.

Table 12: SHA-2 Series Subclasses

Abstract Class	Sub Classes
SHA256	SHA256Cng, SHA256CryptoServiceProvider, SHA256Managed
SHA384	SHA384Cng, SHA384CryptoServiceProvider, SHA2384Managed
SHA512	SHA512Cng, SHA512CryptoServiceProvider, SHA512Managed

CryptoServiceProvider, Cng, and Managed implementations: What's the difference? *.NET algorithms are implemented via a particular API. CryptoServiceProvider implementations wrap functionality provided by the native Win32 CryptoAPI. Cng (Cryptography Next Generation) uses the Cryptography Next Generation API. Managed versions are fully* *implemented in managed (.NET) code. Despite these differences, the output of the algorithms will not (should not) vary.*

The HashAlgorithm Abstract Class

All .NET implementations of hash algorithms implement the **HashAlgorithm** abstract class. This level of abstraction makes it easy and predictable to learn algorithm functionality and write generic solutions. The basic functionality shown in this section can be applied to all of the .NET hash algorithms.

Instances of hash algorithm classes can be created directly (using the **new** keyword), or by using the factory style static **Create** method of the **HashAlgorithm** class and optionally specifying the hash algorithm name as a parameter (**SHA1Managed** is an exception and must be implemented directly). The base class name must be used if supplying an algorithm name to the **Create** method: SHA256, MD5, SHA384, etc. A managed implementation of the specified algorithm will be created unless it's "SHA1", which will create an instance of **SHA1CryptoServiceProvider** (at the time of this writing this is the current default for the **Create** method).

Below, **SHA256Managed** is created directly using the **new** keyword (preferred):

```
SHA256Managed sha = new SHA256Managed();
```

Alternatively, an instance of **SHA256Managed** is created by supplying **SHA256** as a parameter to the **Create** method:

```
HashAlgorithm hash = HashAlgorithm.Create("SHA256");
```

Computing a Hash

The **ComputeHash** method can be used to generate hashes in a manner that will work for almost all scenarios. This method takes a variable-length byte array to hash and returns a fixed-length byte array as the hash result. Overloads allow use of a byte array that specifies a particular index range to hash, or alternatively a **Stream**.

HashAlgorithm implements the **IDisposable** interface; thus, the **using** keyword can be used to provide a deterministic mechanism for freeing the resources that instances consume. Below, **SHA256Managed** is used to return the hash of a byte array:

```
byte[] inputData = new byte[] { 0, 1, 2, 3 };

byte[] hashedData;

using (SHA256Managed sha = new SHA256Managed())
{
    hashedData = sha.ComputeHash(inputData);
}
```

Hashing Data in a Stream

The **ComputeHash** method can also be used with **Stream** objects very easily:

```
MemoryStream m = new MemoryStream(Encoding.UTF8.GetBytes("Stream Data"));

SHA256Managed sha = new SHA256Managed();

byte[] hash = sha.ComputeHash(m);
```

The above example is just for simplicity. Remember to use the **using** statement with your streams to handle cleanup.

Using a CryptoStream

HashAlgorithm also works with **CryptoStream** objects through the **ICryptoTransform** functionality. **CryptoStreams** can be used to process multiple operations and write to an underlying **Stream** object such as a **MemoryStream**. (**CryptoStream** objects were introduced on page 31, and are covered more on page 86)

Below we write some UTF8 encoded bytes to a **CryptoStream**. Once the data has been written to the **CryptoStream** object (be it incrementally or all at once) we can call the **FlushFinalBlock** method to make the **SHA256Managed** object process the hash.

```
byte[] data = Encoding.UTF8.GetBytes("Data to hash");

using (SHA256Managed sha = new SHA256Managed())
{
    using (MemoryStream memStream = new MemoryStream())
    using (var cryptoStream = new CryptoStream(memStream, sha, CryptoStreamMode.Write))
    {
        cryptoStream.Write(data, 0, data.Length);

        cryptoStream.FlushFinalBlock();
        cryptoStream.Close();
        memStream.Close();
    }

    byte[] hash = sha.Hash;
}
```

Verifying a Hash

Verifying or checking a hash is the general process performed to compare an existing hash against a computed hash. If the two hashes match, it is assumed that the input data used to compute the original (existing) hash is the same as the input data for the hash in question.

An example will provide some clarity. A file is hashed. This hash is a fingerprint of the file’s current state. This is stored in some type of persistent storage, or in some cases stored with the file itself. If a question is raised about whether the file has been modified, a second subsequent hash can be performed on the file and compared to the original hash. If the two hashes match, it’s assumed that the files have not been manipulated in any way—they are the same. However, if even a single bit in the file was changed, the hashes will not match and the verification will fail.

Since most hashes are handled as byte arrays, a byte array comparison function will be capable of comparing two hashes. Below is a very common example of a generic byte array comparison function: (we will address the problems that could come from using this function and how it can be improved shortly)

```
bool Compare(byte[] a, byte[] b)
{
    if (a.Length != b.Length) return false;

    for (int i = 0; i < a.Length; i++)
    {
        if (a[i] != b[i]) return false;
    }
    return true;
}
```

After generating a couple of hashes we could use the function as follows:

```
byte[] data = Encoding.UTF8.GetBytes("Data");

SHA256 sha256 = new SHA256Managed();

byte[] firstHash = sha256.ComputeHash(data);

byte[] secondHash = sha256.ComputeHash(data);

bool match = Compare(firstHash, secondHash);
```

There are a couple of details that need to be addressed when using comparison functions. The first is good coding practices. One good aspect of the method is some initial input validation. It disallows byte arrays that don't have matching lengths. This actually prevents a serious security issue: if an attacker can funnel a byte array into a method with a length of one, and that length is used as the variable to loop on, the attacker could force the loop to only render a comparison on one byte. This means that it could only take a maximum of 256 attempts to get a match on the first byte and get the method to return true.

The second issue we need to worry about is whether the function renders a flat comparison time regardless of whether the hash is correct. This technique can prevent timing oracle attacks that seek to capitalize on variances in the processing time of comparison functions, which can indicate how much of a hash was correct. Table 13 shows a sequence of bytes assumed to be a secret hash (your hash) and the hash you are checking against (the attacker's hash). If the time it takes to compare the two sequences is reliable enough, the attacker could determine how many sequences of his hash are correct, make changes until the process takes longer (meaning that he has got more of the sequence correct), and continue to do this until he has discovered the entire hash.

Table 13: Using a timing/verification oracle to map byte sequences in a hash

Your Hash	255	96	5	72	61	88	156	107	201
Attacker's Hash	255	96	5	72	14	111	50	28	200

Success under this type of attack model will of course depend on a number of conditions being present. But it can and does happen. We can overcome the root of the issue—the comparison function—by using a flat comparison function that does not return false until the entire sequence has been checked.

Let's start by looking at an example we've seen floating around in some popular message boards. It uses an integer counter to count the number of discrepancies found in the process and returns true if there are no discrepancies. Can you see a possible problem with this method?

```
bool BadFlatCompare(byte[] a, byte[] b)
{
    int mismatch=0;

    if (a.Length != b.Length) mismatch++;
    for (int i = 0; i < a.Length; i++)
    {
        if (a[i] != b[i]) mismatch++;
    }

    return (mismatch == 0);
}
```

This method will only work correctly if the hashes are the same length. Because argument *a*'s length is what is being looped on, *b* must be the same length or longer to not throw an error. In other words, if argument *a* is longer than argument *b*, an error will result every time. This is not the functionality we want from a comparison method. We would have been better off with a method that produced timing leakage but didn't give us errors.

We could improve upon this slightly if we return false immediately where we get null arguments or if the argument lengths do not match. This level of input validation is reasonable in most cases. The integer counter will still help check the full sequence, but now we don't have to worry about null arrays, or arrays of varying length:

```
bool FlatCompare(byte[] a, byte[] b)
{
    int mismatch = 0;

    if (a == null || b == null) return false;
    if (a.Length != b.Length) return false;

    for (int i = 0; i < a.Length; i++)
    {
        if (a[i] != b[i]) mismatch++;
    }

    return (mismatch == 0);
}
```

Another option is less along the generic lines and would involve performing the hashing of the input as well as the comparison. This is a less trusting style of handling hash comparisons. By using generics, we can check whether our hash size is actually the correct length according to the algorithm we are using. Null input or bad lengths will still be subject to an immediate return of false. After the initial input validation has been performed, the function will run over the entire sequence of both arrays being compared.

```
public bool Check<T>(byte[] hash, byte[] plaintext) where T: HashAlgorithm, new()
{
    if (hash == null) return false;
    if (plaintext == null) return false;

    byte[] computedHash;

    using (HashAlgorithm alg = new T())
    {
        if (hash.Length != alg.HashSize >> 3) return false;

        computedHash = alg.ComputeHash(plaintext);

        alg.Clear();
    }
    int mismatch = 0;

    for (int i = 0; i < computedHash.Length; i++)
    {
        if (computedHash[i] != hash[i])
        {
            mismatch++;
        }
    }

    return mismatch == 0;
}
```

SHA-512/X

We've already seen that the SHA family can be used to produce cryptographic hashes. For instance, when we want a 256-bit hash, we can use SHA-256. However, we can also truncate the output of a hash algorithm like SHA-512 down to the size that we want. This is referred to generally as SHA-512/X (e.g., SHA-512/256, if the output has been truncated to 256 bits). A 256-bit hash derived (truncated) from SHA-512 is actually stronger than a SHA-256 hash (also 256 bits, obviously).

SHA-256 has a second preimage resistance strength between 201 and 256 bits. We can get the full 256-bit strength, however, if we use SHA-512/256, as shown in Table 14. The collision resistance and preimage resistance, however, will not be affected.

Table 14: SHA-512/X strength compared to standard SHA-2 series algorithms

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512	SHA-512/224	SHA-512/256
Collison Resistance Strength in bits	<80	112	128	192	256	112	128
Preimage Resistance Strength in bits	160	224	256	384	512	224	256
Second Preimage Resistance Strength in bits	105-160	201-224	201-256	384	394-512	224	256

Coding a SHA-512/X is very straightforward. The data is hashed with SHA-512 and the result is truncated down to the appropriate length. Below, we have a simple **SHA-512/256** method that uses the **Take** method from **IEnumerable** to take 32 bytes (256 bits) from the beginning of the SHA-512 hash sequence:

```
byte[] Sha512_256(byte[] data)
{
    using(SHA512Cng sha512 = new SHA512Cng())
    {
        return sha512.ComputeHash(data).Take(32).ToArray();
    }
}
```

Example: Hashing with Generics

This example uses generics to simplify computing and verifying hashes, and builds on what we've learned so far in the chapter. Two methods, *ComputeHash<T>* and *VerifyHash<T>*, use a generic type parameter to specify a concrete hash algorithm type, such as SHA256Managed, or SHA512Cng. Developers benefit from the generic behavior because they can use the same methods with different hash algorithms.

ComputeHash<T> takes the plaintext as a parameter and returns the hash:

```
public byte[] ComputeHash<T>(byte[] plaintext) where T : HashAlgorithm, new()
{
    using (HashAlgorithm alg = new T())
    {
        byte[] computedHash = alg.ComputeHash(plaintext);

        alg.Clear();

        return computedHash;
    }
}
```

CheckHash<T> specifies the existing/received hash and the plaintext message as parameters. The plaintext is hashed and compared to the received hash, and the result is returned as bool.

```
public bool CheckHash<T>(byte[] hash, byte[] plaintext) where T : HashAlgorithm, new()
{
    if (hash == null) return false;
    if (plaintext == null) return false;

    byte[] computedHash;

    using (HashAlgorithm alg = new T())
    {
        if (hash.Length != alg.HashSize >> 3) return false;

        computedHash = alg.ComputeHash(plaintext);

        alg.Clear();
    }

    int mismatch = 0;

    for (int i = 0; i < computedHash.Length; i++)
    {
        if (computedHash[i] != hash[i])
        {
            mismatch++;
        }
    }

    return mismatch == 0;
}
```

Example: Hashing with Generics and Strings

This example uses the generic aspect of the last example and also handles plaintext and hash values as strings. To exemplify another method of hash comparison, this verification method uses the string equality operator to determine if the Base64 string hashes match. Keep in mind, however, that this style of comparison may be susceptible to timing attacks.

```
public string ComputeHash<T>(string plaintext) where T : HashAlgorithm, new()
{
    byte[] plaintextBytes = Encoding.UTF8.GetBytes(plaintext);

    using (HashAlgorithm alg = new T())
    {
        byte[] computedHash = alg.ComputeHash(plaintextBytes);

        alg.Clear();

        return Convert.ToBase64String(computedHash);
    }
}
```

```
public bool CheckHash<T>(string hash, string plaintext) where T : HashAlgorithm, new()
{
    if (hash == null) return false;
    if (plaintext == null) return false;

    byte[] plaintextBytes = Encoding.UTF8.GetBytes(plaintext);

    string computedHash=null;

    using (HashAlgorithm alg = new T())
    {
        computedHash = Convert.ToBase64String(alg.ComputeHash(plaintextBytes));

        alg.Clear();
    }

    return (hash == computedHash);
}
```

The code below shows how these methods are used with SHA256Managed:

```
string hash = ComputeHash<SHA256Managed>("Hi!");

bool isMatch = CheckHash<SHA256Managed>(hash, "Hi!");
```

Salting: Defending Against Rainbow Tables

A cryptographic hash function should always produce the same output given the same input. Rainbow tables exploit this predictability. A rainbow table is essentially a lookup of hash values and their corresponding plaintexts given a particular cryptographic hash algorithm. Attackers turn to rainbow tables where they have obtained password hashes and are trying to find the original plaintext. These are most effective against password hashes derived from a low entropy password. If the password was weak, there's a good chance it will be in the table. Rainbow tables can be constructed by an attacker around a set of constraints or downloaded to use with cracking software like John the Ripper.

Most tables are computed around a set of specs that include max password length and complexity (numbers, symbols). Therefore, a rainbow table could be produced for SHA-256 that includes all possible passwords under 8 characters in ASCII. In Table 15, the password hash "6XDEoHE+6457XtB1zc3xx0VKd4g4HdjhymFEsiEVRmk=" could be run against the mock rainbow table to find that the plaintext password is "32max!"

Table 15: Example SHA256 rainbow table

SHA256 Hash Value	Plaintext Password
NlFaMi795BShmRBI2kS8ZWI8jhwx+MMMZSru4FQowjc=	Cat123
6XDEoHE+6457XtB1zc3xx0VKd4g4HdjhymFEsiEVRmk=	32max!
A6xnQhbz4Vx2HuGl4lXwZ5U2I8iziLRFnhP5eNfIRvQ=	1234
XohImNooBHFR0OVvjcYpJ3NgPQ1qq73WKhHvch0VQtg=	password
RJ9+mIT4XzBhheEgtrNcwluHEJ1AqDKpU2gmPgCwAmU=	seahawks

The real benefit to an attacker in using a rainbow table is the speed. Instead of performing a brute force attack or a dictionary style attack, both of which require a mass upfront expenditure of CPU cycles, an attacker can turn to the optimization of indexed databases to find precomputed hashes. You can expect an attacker to find your weak password in a rainbow table within seconds or milliseconds.

So, how can we protect passwords and other low entropy data against rainbow tables? We need to get a bit *salty*.

What is Salting?

In the simplest terms, a salt is a piece (bits) of data that is attached to a password (or other low entropy data) before it is hashed. Most salts are randomly generated sequences of bytes of a particular length, such as 16 bytes (128 bits). Salts are added to secret data so when it's passed through a cryptographic function, the derivative is a product of the secret data and the salt, not just the secret data. This helps thwart attacks like rainbow tables that depend on the predictability of certain plaintext values producing certain hashes. Typically, salts do not need to be kept secret.

From an attacker's point of view, the salt has destroyed the efficiency of precomputed cracking resources like rainbow tables. The work to crack the password must now incorporate the salt and occur upfront. This considerably changes the time it takes to compromise even a weak password hash. While this will not actually stop the attacker from performing the work to crack the password, the work factor might be seen as infeasible due to cost or current resources and technologies. Random salting will also prevent identical user passwords from rendering the same hashes.

How Does Salting Work?

Normally, a hash would be computed as in the example below, where *h* is the hash algorithm, *m* is the plaintext message, and *x* is the function's output:

x = h(m)

To add a salt to the function above, the salt *s,* could be randomly generated and prepended to the message *m*:

s = cryptographically random material

x = h(s+m)

Salt Storage

Salts must be stored *somewhere* because they are needed to successfully compute and compare hashes to the stored hash on record. Salts don't need to be kept secret, but this is at the developer's discretion.

The most common method of storing a salt is to attach it to the hash itself. This is convenient because the salt is stored with the data it pertains to and no other lookups are required to obtain the salt. In this scenario, if the hashes are compromised the salts will be compromised, too. But the attacker will still have to do all of the cracking upfront.

The other, less common method of storing the salts is to store each salt in a secured database or lookup. The salt is usually linked to the user or object via a foreign key or ID. In this scenario, to compute a hash and compare it to the original, the salt would need to be retrieved from its secure storage before this verification could take place.

Some solutions have opted not to store randomly generated salt data, and instead turn to static peripheral data like a user name or account ID to use as salt. While semi-effective, this is not a best practice and not considered as secure as cryptographically random salts.

Chapter 5 also covers aspects of salting and salt storage.

Hash Generation and Verification using Salts

Last section we explained the principles behind salting. In this example you will learn how to hash and verify salted data. We'll start with salt generation.

Generation

Salts should be cryptographically random and at least the length of your hash. Data hashed with SHA-256, for example, should use 256-bit salts. Each piece of data being hashed—each hash operation—should be given its own random salt. **GetBytes** in **RNGCryptoServiceProvider** is used to fill a byte array with a cryptographically secure random sequence, making it a good candidate for secure salt generation. Do not use the **Random** class. Generating a 256-bit salt can be done like so:

```
byte[] salt = new byte[32];

RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

rng.GetBytes(salt);
```

The salt will need to be prepended to the data prior to being hashed. There are a few ways to do this, the most common being the **Array.Copy** method, or the **Concat** method in **IEnumerable**. We will use the latter for its intuitive interface and simplicity. Notice that we have to call the **ToArray** method after **Concat**:

```
byte[] data = Encoding.UTF8.GetBytes("weak password");
```

```
byte[] saltAndData=salt.Concat(data).ToArray();
```

The concatenated salt and data can now be passed into the hash function, in this case the **ComputeHash** method of the **SHA256Cng** class. The resulting hash is now a product of the salt as well as the data:

```
SHA256Cng sha256 = new SHA256Cng();

byte[] hash = sha256.ComputeHash(saltAndData);
```

Remember, we need to keep track of the salt in order to correctly verify the hash. So, we'll concatenate the salt and the hash:

```
byte[] saltAndHash = salt.Concat(hash).ToArray();
```

Verification

To verify the hash, we can use the same process—the same method in fact—that we used to render the original hash. Positive verification results from the computed hash matching the stored/original hash.

First, we need the salt. If the salt is stored with the data, as our example thus far has demonstrated, we can take the portion of the data containing the salt. If we are handling salts and data in byte array format, we will need to know how much of the array contains the salt. This is easy in our example because we know the length of the salt.

```
byte[] foundSalt = saltedHash.Take(32).ToArray();
```

Once we have the salt, we can concatenate the salt with the input data, and hash it like we did before:

```
byte[] inputData = Encoding.UTF8.GetBytes("weak password");

byte[] foundSaltAndInputData = foundSalt.Concat(inputData).ToArray();

byte[] computedHash = sha256.ComputeHash(foundSaltAndInputData);
```

If our computed hash matches the stored hash, we have a positive verification. The same rules for verification/comparison methods apply from the *Verifying a Hash* section.

Example: Salted Hashing and Verification

The following two methods incorporate what we've covered in the last sections. The first, *SaltedSHA256*, handles salt generation, hashing, and attaching the salt to the hash.

```
byte[] SaltedSHA256(byte[] plaintext)
{
    byte[] salt = new byte[32];

    using (RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider())
    {
        rng.GetBytes(salt);
    }

    using (SHA256Cng sha256 = new SHA256Cng())
    {
        byte[] saltAndPlaintext = salt.Concat(plaintext).ToArray();
        byte[] hash = sha256.ComputeHash(saltAndPlaintext);
        return salt.Concat(hash).ToArray();
    }
}
```

The second method, *VerifySaltedSHA256*, returns a bool value for verification. It is made to work with the *SaltedSHA256* method and verify data in that format. Therefore, it is not suited for generic comparison/verification purposes. However, you'll notice we still deliver a flat verification time if the input is formatted correctly.

```
bool VerifySaltedSHA256(byte[] saltedHash, byte[] plaintext)
{
    if (saltedHash == null) return false;
    if (plaintext == null) return false;

    byte[] salt = saltedHash.Take(32).ToArray();
    byte[] foundHash = saltedHash.Skip(32).ToArray();

    byte[] computedHash;

    using (SHA256Cng sha256 = new SHA256Cng())
    {
        if (foundHash.Length != sha256.HashSize >> 3) return false;

        byte[] saltAndPlaintext = salt.Concat(plaintext).ToArray();

        computedHash = sha256.ComputeHash(saltAndPlaintext);

        sha256.Clear();
    }

    int mismatch = 0;

    for (int i = 0; i < computedHash.Length; i++)
    {
        if (computedHash[i] != foundHash[i])
        {
            mismatch++;
        }
    }

    return mismatch == 0;

}
```

The methods can be used to hash and verify data easily:

```
byte[] password = Encoding.UTF8.GetBytes("myPassword");

byte[] hash = SaltedSHA256(password);

byte[] inputPassword =Encoding.UTF8.GetBytes("myPassword");

bool match = VerifySaltedSHA256(hash, inputPassword); //returns true
```

Throughout this example we've used byte-array format to handle all of our data. We do this because so many of the examples in forums and message boards handle data in string format and use identifiers in the string data to split the salt and the hash or other metadata associated with the hashing process. The problem with this approach is that so many of these developers do not know how to concatenate and separate salts when they are working strictly with byte arrays. So, in the event you need to transmit or store data in Base64 format, conversions can be performed pretty easily. Here is our last example using a Base64 conversion in-between:

```
byte[] password = Encoding.UTF8.GetBytes("myPassword");

byte[] hash = SaltedSHA256(password);

//Convert to Base64 string.
string b64Hash = Convert.ToBase64String(hash);

//Transmit or store hash string.
```

Now for the verification:

```
byte[] inputPassword =Encoding.UTF8.GetBytes("myPassword");

//Retreive string hash and convert to byte array.
byte[] originalHash = Convert.FromBase64String(b64Hash);

bool match = VerifySaltedSHA256(originalHash, inputPassword); //returns true
```

Lastly, note that this example showed a simple way to hash and verify data using a salt. There will be instances where you need to interface with existing protocols or standards that use their own formats for how salts and metadata need to be attached, stored, and formatted. In those cases you’ll likely be using the tools you’ve acquired here to meet their specification.

Example: Generic Salted Hash Class

This example combines the generic hash wrapper class example from earlier in the chapter with the last example, salted stretching and verification. We will not bother walking through the intricacies since they have been covered in the other examples. However, a new aspect in this class that is worth noting is the use of the HashAlgorithm instance’s HashSize property. We use this property to correctly determine the salt length that must be generated and subsequently retrieved from the hash. Thus, if a 256-bit hash is being used as the generic type parameter, a 256-bit salt will be generated, and a 512-bit salt for a 512-bit hash.

```
public sealed class SaltedHashWrapper<T> where T : HashAlgorithm, new()
{
    public byte[] ComputeHash(byte[] plaintext)
    {
        using (HashAlgorithm hashAlg = new T())
        {
            int hashByteSize = hashAlg.HashSize>>3;

            byte[] salt = new byte[hashByteSize];

            using (RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
            }

            byte[] saltAndPlaintext = salt.Concat(plaintext).ToArray();
            byte[] hash = hashAlg.ComputeHash(saltAndPlaintext);
            return salt.Concat(hash).ToArray();
        }
    }
    public bool CheckHash(byte[] saltedHash, byte[] plaintext)
    {
        if (saltedHash == null) return false;
        if (plaintext == null) return false;

        byte[] foundHash;
        byte[] computedHash;

        using (HashAlgorithm hashAlg = new T())
        {
            int hashByteSize = hashAlg.HashSize >> 3;

            byte[] salt = saltedHash.Take(hashByteSize).ToArray();
            foundHash = saltedHash.Skip(hashByteSize).ToArray();

            if (foundHash.Length != hashByteSize) return false;

            byte[] saltAndPlaintext = salt.Concat(plaintext).ToArray();

            computedHash = hashAlg.ComputeHash(saltAndPlaintext);

            hashAlg.Clear();
        }

        int mismatch = 0;

        for (int i = 0; i < computedHash.Length; i++)
        {
            if (computedHash[i] != foundHash[i])
            {
                mismatch++;
            }
        }

        return mismatch == 0;
    }
}
```

An instance is created directly by placing a class that derives from **HashAlgorithm** in the generic type parameter:

```
SaltedHashWrapper<SHA256Managed> alg = new SaltedHashWrapper<SHA256Managed>();
```

Here is an example using **SHA384Cng:**

```
byte[] password = Encoding.UTF8.GetBytes("myPassword");

SaltedHashWrapper<SHA384Cng> alg = new SaltedHashWrapper<SHA384Cng>();

byte[] hash = alg.ComputeHash(password);

byte[] inputPassword = Encoding.UTF8.GetBytes("myPassword");

bool match = alg.CheckHash(hash, inputPassword); //returns true
```

And another using **SHA256Managed**:

```
byte[] password = Encoding.UTF8.GetBytes("myPassword");

SaltedHashWrapper<SHA256Managed> alg = new SaltedHashWrapper<SHA256Managed>();

byte[] hash = alg.ComputeHash(password);

byte[] inputPassword = Encoding.UTF8.GetBytes("myPassword");

bool match = alg.CheckHash(hash, inputPassword); //returns true
```

SHA-3

In 2007, NIST announced the public competition for a new cryptographic hash function, SHA-3. The specs called for minimum output lengths of 256 and 512 bits. Five years later, and after much scrutiny, the *Keccak* algorithm was selected as the winner. At the time of this writing, .NET does not currently implement SHA-3. If you want a SHA-3 implementation for .NET, you should look for a third-party library rather than waiting for it to come out in a .NET release.

Recommendations

The primary security issue with hash algorithms is collisions. Collisions occur because an algorithm can have infinite inputs but only 2^n possible outputs, where *n* is the output size in bits. For example, SHA1 creates a 160 bit hash, equating to 2^{160} possible combinations of outputs. But this only gives us 80 bits of collision resistance under a generic attack model. For modern security systems, you should be aiming for a *minimum* of 128 bits of strength. This means that any hash algorithm you use must have *at least* 256 bits of output. In the current .NET framework this only leaves three choices: SHA-256, SHA-384, and SHA-512. SHA-256 is recommended at minimum for 128-bit strength. Conservative recommendations would be for SHA-384. Where true 256-bit security is the goal, SHA-512 is the only option in .NET.

Many developers will have the issue of legacy compliance with insecure algorithms, the most common of which is MD5. In these situations you should always raise the issue to the team lead or project lead (whoever is acting in this capacity) to report the use of insecure algorithms. Another factor is going to be the purpose and context of the hashing. The question to ask here is “what are the consequences of a collision?” Does it mean illegitimate permissions, authentication, or access? There are seldom scenarios where it doesn’t. It is simpler and more secure to default to SHA256 and raise the bar from there if necessary. In terms of classes, the managed implementations are recommended over the CryptoServiceProvider implementations. Thus, our recommendation for 128-bit systems is **SHA256Managed**.

Chapter Summary

- Hash algorithms are one-way functions that provide a deterministic fixed-length output given a variable-length input. They act as a digital fingerprint and are used to verify the integrity of different types of data.
- SHA256 or stronger is recommended for a secure hash algorithm. MD5 and SHA1 should not be used in modern production applications unless there is a legacy compliance issue that specifically requires these algorithms to be used.

- It's a best practice to store the hash of a user password rather than the plaintext password itself. The password is verified when a user enters a password, which is then hashed and compared against the stored password hash.
- Salts are used to defeat rainbow table attacks. Salts should be randomly generated (cryptographically secure, see the **RNGCryptoServiceProvider** class) and be at least the length of the hash algorithm. SHA256 would therefore need a 256-bit salt.
- As a general rule of thumb, the collision strength (in bits) of a cryptographic hash algorithm is half the bit length of the message digest. For example, SHA-256 has a collision strength of 128 bits.

Chapter Questions and Exercises

1. Explain the differences between collision resistance, preimage resistance, and second preimage resistance.
2. Explain why SHA256 is considered a stronger hash algorithm than SHA1. What aspects of SHA256 make it stronger? How is SHA512 stronger than SHA256?
3. What is a rainbow table? How do salts help defend against them?
4. Write a simple program to hash and verify input data.
5. Write a simple rainbow table for 4 digit numeric hashes and test it. This is small enough to do in-memory, however, you could also use a database for a lookup.

Scenarios

1. You are developing a web application where users create accounts based on a set of credentials: email and password. And you know that you need to store a hash of the password rather than the plaintext password. What might be considerations for selecting a hash algorithm?
2. Building on scenario 1, suppose you are worried about password hashes being compromised by a rainbow table and wisely decide to salt your user passwords. What size salt should you use and how should it be stored? What issues might crop up in terms of robustness and flexibility of your solution. How could these be prevented?
3. You need to build a solution that hashes a file for the purposes of ensuring its integrity. The file will usually be 20-30MB in size. Your project leader recommends that you use a salt in the hashing process, but still asks for your opinion. What should you tell her? Is the salt necessary? Why or why not?

5 Password-Based Key Derivation Functions

Password-Based Key Derivation Function: *A function that, with the input of a cryptographic key or shared secret, and possibly other data, generates a binary string, called keying material.*

Chapter Objectives

1. Understand the purpose and general uses of PBKDFs.
2. Learn how to securely generate and manage salts.
3. Understand the relationship between stretching iterations and effective key strength.
4. Learn how to derive keys from string passwords using the PBKDFs in .NET and how they could be implemented in an application.

When a user's data needs to be protected, it's convenient for them to use a password or a passphrase rather than a key. There are two main problems with this. First, this password is destined to be weak, which practically defeats the purpose of using a strong encryption algorithm. Second, the password is probably not going to match the requisite size for the encryption key. There is an off chance that a user might use a password that is the right size, but what about the rest of the users? We need a mechanism to derive a fixed-length key from a weak, variable-length password.

A regular hash function could do this, right? Yes, it could. In fact, as you may recall, the SHA-2 family was built specifically to produce hashes that were exactly twice the size of AES keys. However, dictionary attacks, rainbow tables, and crackers such as John the Ripper*,* are still effective against hashes derived from weak passwords.

There is a better option. Password Based Key Derivation Functions (PBKDFs) are used to derive a cryptographically secure key from a low entropy password. Like hash algorithms, PBKDFs are deterministic, meaning they will derive the same key given the same input, every time. These functions offer advantages over deriving a key with a regular cryptographic hash algorithm. The first advantage is that most implementations of PBKDF functions derive a key of arbitrary length. This means that you can use these functions to derive a perfectly sized key for a symmetric key algorithm. Second, they make it easy to supply salts and stretching iterations (the number of times the data is cycled/hashed before a key is returned). This crushes the rainbow table attack. It also gives the developer a reasonable assurance that their function will consume a predefined number of CPU cycles (time).

Salting and Stretching

Salting and Stretching helps increase password and key security in two ways: the use of salts defeats the effectiveness of rainbow tables by adding unpredictable data; stretching adds additional computations (the stretching iterations) to derive a key, which adds to requisite workload placed on an attacker. This means that any attack that uses the stretching function must do this extra work, resulting in an increase in effective key size.

What is a stretching function? *A stretching function is a function that takes a key/password and a salt and/or a number of stretching iterations as parameters, and returns a derived key. Developers are left with the choice of writing their own stretching function or using one of the standardized library implementations available. It's always recommended to use the library approach because it's more productive and usually more secure.*

To understand how salting and stretching can increase effective key size, we will draw up a theoretical stretching function *H* that derives a key *K,* where we input a salt *s,* a password *p*, and specify the number of iterations, *r*:

$$K=H(s,p,r)$$

The increase in key size is realized where the attacker must perform *r* computations in the stretching function. If $r=2^{10}$, the attacker must perform an additional 2^{10} computations for each password that is tried. Therefore, trying 2^{100} passwords would take 2^{110} hash computations, making the effective key size 10 bits longer.

Stretching Function Execution Time: *As a best practice, stretching functions should take at least 200 milliseconds to execute (200-1000 milliseconds is the recommended range). Remember, this refers to the algorithm's stretching time, not peripheral tasks associated with the function such as generating salts.*

Moore's Law says that number of transistors on integrated circuits double *approximately* every two years. This means that over time, data derived from cryptographic functions will be attacked with increasing speed/resources, leading to a reduced mean-time-to-compromise (MTTC). To compensate for this, developers can increase the computations in their stretching functions to offset the gains in computing speed.

Rfc2898DeriveBytes and PasswordDeriveBytes

The Public Key Cryptography Standard #5 (PKCS5) outlines key derivation functions that incorporate salting and stretching processes; namely Password Based Key Derivation Function 1 and 2 (PBKDF1 and PBKDF2). Both can be used to produce derived keys, and salt and stretch passwords. PBKDFs are regularly implemented by security libraries. The **System.Security.Cryptography** offers developers the **PasswordDeriveBytes** class (PBKDF1) and **Rfc2898DeriveBytes** (PBKDF2).

Both **PasswordDeriveBytes** and **Rfc2898DeriveBytes** implement the **DeriveBytes** abstract class. This class defines two public abstract methods:

- **GetBytes(int):byte[]** This method returns a byte array of the specified size, the content of which is a result of the constructor information and algorithm of the derived class.
- **Reset()** This resets the algorithm to its original constructed state.

The **GetBytes** method is used heavily by both classes and is the primary means of obtaining the derived byte arrays.

Rfc2898DeriveBytes (PBKDF2)

Rfc2898DeriveBytes is the go-to class for PBKDF functionality and general-purpose salting and stretching. At a minimum, the constructor must be supplied a password and a salt. Overloads also allow the number of stretch iterations to be supplied.

An Rfc2898DeriveBytes class can be directly created like this:

```
Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(password, salt, iterations);
```

The **GetBytes** method can then be used to derive a specified number of bytes. Below we derive 32 bytes of data:

```
byte[] derivedBytes = pbkdf2.GetBytes(32);
```

A more realistic and complete example is shown below with a random 256-bit salt:

```
byte[] passwordBytes = Encoding.UTF8.GetBytes("password");

byte[] salt = new byte[32];

new RNGCryptoServiceProvider().GetBytes(salt);

int iterations = 10000;

Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(passwordBytes,salt,iterations);

byte[] derivedBytes = pbkdf2.GetBytes(32);
```

Salt Sizes and Generation: *Salts should be randomly generated using a secure generator like the RNGCryptoServiceProvider class and be at least the length of the hash being produced. If the hash is 256-bit, the salts should also be 256-bit. Most examples in this chapter use 256-bit (32-byte) hashes and 256-bit salts.*

Example: Deriving and Verifying Stretched Passwords

Abstracting away the salting and iteration details behind a simple *DerivePassword* method provides encapsulation and a cleaner interface without the messy details. In this case the derived key is prepended with the salt so it can be used in a verification process (covered next):

```
byte[] DerivePassword(string password)
{
    byte[] passwordBytes = Encoding.UTF8.GetBytes(password);

    byte[] salt = new byte[32];

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(salt);

    byte[] derivedBytes;

    using (var pbkdf2 = new Rfc2898DeriveBytes(passwordBytes, salt, 10000))
        derivedBytes = pbkdf2.GetBytes(32);

    return salt.Concat(derivedBytes).ToArray();
}
```

The verification method takes the derived password from the *DerivePassword* method and the plaintext password to be verified and returns a bool result. Inside the method, the salt is removed from the derived password bytes and used to stretch the supplied plaintext password. The computed password and the received stretched password are then compared byte-by-byte and the bool result is returned indicating the outcome of the verification:

```
bool VerifyPassword(byte[] derivedPassword, string plaintextPassword)
{
    byte[] salt = derivedPassword.Take(32).ToArray();

    byte[] derivedBytes = derivedPassword.Skip(32).ToArray();

    byte[] passwordBytes = Encoding.UTF8.GetBytes(plaintextPassword);

    byte[] computedBytes;

    using (var pbkdf2 = new Rfc2898DeriveBytes(passwordBytes, salt, 10000))
        computedBytes = pbkdf2.GetBytes(32);

    if (computedBytes.Length != derivedBytes.Length) return false;

    int mismatch = 0;

    for (int i = 0; i < derivedBytes.Length; i++)
    {
        if (computedBytes[i] != derivedBytes[i]) mismatch++;
    }
    return mismatch == 0;
}
```

In the above examples we hardcoded our stretch iterations at 10000 in both the *DerivePassword* and *VerifyPassword* methods. The verification method will not be able to correctly determine a match if it doesn't use the same number of stretch iterations as the derivation method. Remember to keep this in mind if you ever decide to increase or decrease stretch iterations on a production system. It will be a breaking change unless the stretch iterations are also stored along with the salt/data and can be dynamically adjusted (we will address this issue a little later in the chapter).

Example: Deriving Keys for Symmetric Encryption

Our last example showed how you can use **Rfc2898DeriveBytes** (PBKDF2) to create highly random stretched passwords, and how they are verified. PBKDF2 is also commonly used to derive a key of a particular length. The derivation process is essentially the same as our last example: derive a key from a salt and a specified number of iterations. Verification, however, will be replaced with decryption. The key will still have to be reproduced with the correct salt and number of iterations. Last example, we stored the salt with the hash so it could be used in the verification process. In this case we don't have a hash to store our salt with. Instead, we will attach the salt to the ciphertext (the output from the encryption function). Symmetric encryption is covered next chapter, so we'll call generic *Encrypt* and *Decrypt* methods for now (refer to page 89 for the code for these methods):

```
byte[] EncryptWithPassword(string password, byte[] data)
{
    byte[] passwordBytes = Encoding.UTF8.GetBytes(password);

    byte[] salt = new byte[32];

    byte[] derivedKey;

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(salt);

    using (Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(passwordBytes, salt, 10000))
    {
        derivedKey = pbkdf2.GetBytes(32);
    }

    byte[] ciphertext = Encrypt(data, derivedKey);

    return salt.Concat(ciphertext).ToArray();
}
```

To decrypt, the salt must be removed to derive the symmetric key using the password:

```
byte[] DecryptWithPassword(string password, byte[] data)
{
    byte[] passwordBytes = Encoding.UTF8.GetBytes(password);

    byte[] salt = data.Take(32).ToArray();

    byte[] ciphertext = data.Skip(32).ToArray();

    byte[] derivedKey;
    using (Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(passwordBytes, salt, 10000))
    {
        derivedKey = pbkdf2.GetBytes(32);
    }

    return Decrypt(ciphertext, derivedKey);
}
```

Their usage is straightforward:

```
string password = //set your password

byte[] data = new byte[100];

byte[] cipher = EncryptWithPassword(password, data);

byte[] plaintext = DecryptWithPassword(password, cipher);
```

PasswordDeriveBytes (PBKDF1)

The **PasswordDeriveBytes** class implements PBKDF1 and shouldn't be used except for legacy compatibility.

PasswordDeriveBytes will salt and stretch a key or password and functions like the PBKDF2-based **Rfc2898DeriveBytes** class. One difference is that the hash algorithm must be specified in **PasswordDeriveBytes** if you want the object to use a hashing algorithm other than **SHA1**. This is shown below in our example where we have specified "SHA256" in the **PasswordDeriveBytes** instance constructor.

```
byte[] password = Encoding.UTF8.GetBytes("password");

byte[] salt = new byte[32];

int iterations = 10000;

using (var rng = new RNGCryptoServiceProvider())
    rng.GetBytes(salt);

byte[] stretchedPassword;

using (var pbkdf1 = new PasswordDeriveBytes(password, salt, "SHA256", iterations))
    stretchedPassword = pbkdf1.GetBytes(32);
```

Below are two functions that encapsulate the gritty details of salting and stretching and provide a simple interface to derive a password and verify an existing stretched password against a received plaintext password (these have the same functionality as the PBKDF2 examples covered last section):

```
byte[] DerivePasswordPBKDF1(string password)
{
    byte[] passwordBytes = Encoding.UTF8.GetBytes(password);

    byte[] salt = new byte[32];

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(salt);

    byte[] derivedPassword;

    using (var pbkdf1 = new PasswordDeriveBytes(passwordBytes, salt, "SHA256", 10000))
```

```
        derivedPassword = pbkdf1.GetBytes(32);

    return salt.Concat(derivedPassword).ToArray();
}
```

```
bool VerifyPasswordPBKDF1(byte[] derivedPassword, string plaintextPassword)
{
    byte[] salt = derivedPassword.Take(32).ToArray();

    byte[] derivedBytes = derivedPassword.Skip(32).ToArray();

    byte[] passwordBytes = Encoding.UTF8.GetBytes(plaintextPassword);

    byte[] computedPassword;

    using (var pbkdf1 = new PasswordDeriveBytes(passwordBytes, salt, "SHA256", 10000))
        computedPassword = pbkdf1.GetBytes(32);

    if (computedPassword.Length != derivedBytes.Length) return false;

    int mismatch = 0;

    for (int i = 0; i < derivedBytes.Length; i++)
    {
        if (computedPassword[i] != derivedBytes[i]) mismatch++;
    }

    return mismatch == 0;
}
```

Security Issue: *You should never ask for more bytes than the PasswordDeriveBytes.HashName algorithm can provide (e.g. 20 bytes maximum for SHA-1, 32 for SHA256).*

Context and Workload: Do We Care about Iterations or Time?

Most stretching functions can be given a number of iterations to execute. And in some cases we are actually concerned about the *number* of iterations, but not usually. Usually, we care about the *time* it takes for the specified number of iterations to execute. The more iterations the attacker has to compute, the more *time* it takes her; and the less effective her attacks are. Best practices call for a stretching function to run between 200 and 1000 milliseconds. This is a simple concept. But how are we expected to know how long our stretching functions take? As you can imagine, it’s going to take testing.

So, if what we actually care about is the execution time of our function, then why don’t we tell it how *long* to run instead of how many iterations to run? Meaning that we could tell the function to run for 200 milliseconds and however many iterations it computes in that time period is what it will use. Well, for starters, it’s less deterministic. If the function is told to run for 200 milliseconds, it will do so no matter its environment or platform. The problem here is that a slow environment might only run 1,000 iterations (this is an arbitrary number). While a performant environment might run 9,000 iterations in the same amount of time. Such disparity is bad when you’re trying to establish a baseline of security.

This comes back to context and what we are expecting from the application. In a server environment, we might expect the server to actually be performing the work for the attacker through requests. It's a different story if the attacker possesses the hash and can attack it at leisure. A series of parallel GPUs is obviously going to blow away the number of iterations our regular server executes in 200 milliseconds. One second of work in our world could be a mere millisecond in theirs.

You've seen that you can use stretching iterations to increase effective key size. You've also seen that we care a lot about time, but that we can't consider workload associated with time to be a constant thing. So, what then are we supposed to do? Ultimately, all we can do is stick to best practices and make sure we have a firm understanding of our context. As you may have guessed, we need to be able to adjust our stretching functions so they remain secure given increases in technology and changes in context.

Safely Adjusting Stretch Iterations in a Production Environment

Sometimes stretch iterations need to be increased as a countermeasure. At the beginning of this book we mentioned that the future is here but the past is everywhere around us. This is important when considering the effectiveness of stretching functions.

How about a simple example. Let's say a program was written 10 years ago that used a stretching function to thwart attacks against user passwords. And let's assume that this function took .1 seconds to run. In simple terms we'll assume the attacker could make at most 10 attempts per second. Now assume that today, given the increased technologies and processing power, this same function only takes .04 seconds. An attacker could now compute over twice the number of passwords. In other words, his attack is over twice as efficient.

The problem is that by the time we recognize the trend and need to adjust the stretching iterations, we already have a production system on our hands that has secured countless user passwords with this now questionable algorithm. This is a bad position to be in and one that many organizations and developers find themselves facing. To clarify, we'll define our problem as not being able to increase the existing stretching iterations on passwords or keys that are *currently* being salted and stretched. Our problem does not encompass the fact that older passwords may now be vulnerable to today's attacks (which is a valid point). For now we'll just worry about increasing the strength of the new ones without breaking the old ones.

Let's look at a few ways to more flexibly and dynamically stretch and verify input data.

Ensuring Future Security and Compatibility

The critical breakdown in most stretching functions is that verifying previously computed data relies on the algorithm to use the same salt and number of stretching iterations. Overcoming this problem is trivial and is essentially the same process for how salts are usually handled. Simply prepend the data with the number of iterations and have the verification function parse this out and dynamically set its iterations. Essentially, our stretched packets could take on the following colon-separated format if we are persisting the data in base64:

[iterations]:[salt]:[derived key]

Parsing is simple: split the string on the colons and you've got the parts you need. For those maintaining a byte-array-only approach, it's also simple enough to break up the array on predetermined indices. Supplying the algorithm ([algorithm name]:[iterations]:[salt]:[derived key]) would, however, make for a more robust solution over the long term. Make sure you research how your data should be formatted if you are integrating with an existing protocol.

Below, we use the same colon-separated format we described above. The stretching function, *StretchPassword*, takes the plaintext password and the stretch iterations as parameters and returns a string containing the stretch iterations, the salt, and the derived key. The salts and the derived hash are both 256-bit:

```
string StretchPassword(string password, int iterations)
{
    byte[] passwordBytes = Encoding.UTF8.GetBytes(password);

    byte[] salt = new byte[32];

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(salt);

    byte[] derivedBytes;

    using (var pbkdf2 = new Rfc2898DeriveBytes(passwordBytes, salt, iterations))
        derivedBytes = pbkdf2.GetBytes(32);

    string saltString = Convert.ToBase64String(salt);

    string iterationsString = iterations.ToString();

    string derivedString = Convert.ToBase64String(derivedBytes);

    return iterationsString + ":" + saltString + ":" + derivedString;
}
```

The verification function does not contain a parameter for iterations as it will retrieve them from the derived data in the first parameter. The second parameter is for the plaintext password or key:

```
bool VerifyPassword(string derivedData, string plaintextPassword)
{
    string[] parts = derivedData.Split(':');

    int iterations = Convert.ToInt32(parts[0]);

    byte[] salt = Convert.FromBase64String(parts[1]);

    byte[] passwordBytes = Encoding.UTF8.GetBytes(plaintextPassword);

    byte[] computedBytes;

    using (var pbkdf2 = new Rfc2898DeriveBytes(passwordBytes, salt, iterations))
        computedBytes = pbkdf2.GetBytes(32);

    return parts[2] == Convert.ToBase64String(computedBytes);
}
```

Using these methods is as simple as:

```
string stretchedPassword = StretchPassword("hello",1000);

bool ok = VerifyPassword(stretchedPassword, "hello"); //returns true
```

Notice that all of the salting and stretching details besides the number of iterations are encapsulated by the methods and hidden from the user. In most circumstances we want to make our method interfaces as simple as possible to reduce the amount of data we have to maintain and handle. However, in situations where salts are stored separately from the derived key (salt storage alternatives will be covered next section), the interface will obviously need to be opened up to accommodate for the salt data.

*Remember, although the method signatures are simple, their internal functionality increases complexity. And complexity is our biggest enemy.

***Don't Forget Input Validation**: Validation should always be performed If any of the data going into the stretching method is obtained from the user or any other untrusted area.*

Example: IntelliStretch/Time-Centric Stretching

A creative option for using stretching algorithms like PBKDF2 is governing their execution time rather than their iterations: *time-centric* instead of *iteration-centric*. This usually involves encapsulating the function in a loop that exits after a certain timeframe or interval has elapsed. The stretching function's current iteration and the hashed data are then returned once the loop breaks.

This model is ideal for applications that need their stretching function to take a flat amount of time regardless of environment. The benefit of this model is that it guarantees a stretching function will always adhere to best practices (200-1000 milliseconds) even if the app is moved to a faster machine. This has been used as a method of keeping stretching functions running in the *present* so that they can maintain a minimum level of security autonomously. Conversely, *iteration-centric* functions will ensure a fixed number of iterations, but when moved to a faster platform have no way of adjusting themselves and are basically *stuck in the past*.

Here's an example. Joe writes a *time-centric* stretching function for piece of web-based software that will execute for 200 milliseconds. On his current server the function usually executes about a 4,000 iterations in this timeframe. A couple years later Joe buys a Ferrari server. He migrates his software to the new server and can't notice any difference. This is because his *time-centric* function is going to run for 200ms no matter what. Only now, the redlining Ferrari is cranking out 18,000 iterations in the same time that our old one did 4,000.

Below, we have a *time-centric* stretching function, *TimedStretch*, which takes a plaintext password and the milliseconds for the function to run. It returns the iterations, the salt, and the 32-byte hash in the colon-separated format we covered last section:

```
public string TimedStretch(string password, int ms)
{
    byte[] derivedKey = Encoding.UTF8.GetBytes(password);

    byte[] salt = new byte[32];

    int iterations = 0;

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(salt);

    DateTime start = DateTime.Now;

    do
    {
        using (var pbkdf2 = new Rfc2898DeriveBytes(derivedKey, salt, 1000))
            derivedKey = pbkdf2.GetBytes(32);
        iterations += 1000;
    }
    while ((DateTime.Now - start) < new TimeSpan(0, 0, 0, 0, ms));

    return iterations.ToString() + ":" +
            Convert.ToBase64String(salt) + ":" +
            Convert.ToBase64String(derivedKey);
}
```

You'll notice that the above function executes 1,000 stretch iterations per one loop iteration (in this case, the nature of the do/while loop adds for a nice safety feature by always executing a minimum of 1,000 iterations). A **DateTime** object is used to keep track of how much time has elapsed during the loop. The loop breaks upon exceeding the time specified in the method parameter (obviously dependent on a properly functioning **DateTime** class). 256-bit salts are randomly generated and handled internally.

The verification method does not worry about anything to do with time. It does, however, have to follow the same formatting in its loop to execute 1,000 stretching iterations per one loop iteration. If it doesn't, and attempts to execute all of the iterations in a single instance of **Rfc2898DeriveBytes**, it won't verify correctly. Like some of the other verification methods we've written, this one encapsulates all of the parsing detail and only needs the derived hash in the format in which it was produced (colon-separated) and the plaintext password that it must verify:

```
public bool VerifyTimedStretch(string derivedHash, string password)
{
    string[] parts = derivedHash.Split(':');

    int iterations = Convert.ToInt32(parts[0]);

    byte[] salt = Convert.FromBase64String(parts[1]);

    byte[] computedKey = Encoding.UTF8.GetBytes(password);

    for (int i = 0; i < iterations; i += 1000)
    {
        using (Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(computedKey, salt, 1000))
            computedKey = pbkdf2.GetBytes(32);
    }

    return (parts[2] == Convert.ToBase64String(computedKey));
}
```

Remember, from the front end, the time-centric model is a *nondeterministic* derivation process. What this means is that you may get a different result between passes even though all arguments are the same. This is why the verification method gets the iteration count from the stored hash before it can properly compute and verify a password. If the password was computed first, it could actually run for a different amount of iterations depending on the timer, and not verify correctly. Ultimately, time-centric stretching will work fine for hashing and verification if your verification method computes the hash using the iterations from the stored/original hash like our examples above.

Below, *TimedStretch* will salt and stretch a string, "pass", for 250 milliseconds, and then undergo subsequent verification on the next line. Both methods are designed to be used together:

```
string derivedHash = TimedStretch("pass", 250);

bool ok = VerifyTimedStretch(derivedHash, "pass");
```

This type of stretching model won't be practical for all applications. It will, however, keep stretching function execution times consistent and help account for an application getting moved to a faster platform where the same (previous) work factor could be performed even faster.

Example: PBKDF with SHA256

It's recommended to use a standardized function like PBKDF2 rather than writing your own. Still, you could use a cryptographically secure hash function like SHA256 to perform iterative salting and stretching of a plaintext password. Below, *StretchPasswordSHA256* delivers this functionality and returns a colon-separated string containing the salt, the iterations, and the hash:

```
private string StretchPasswordSHA256(string password, int iterations)
{
    byte[] salt = new byte[32];

    using (var rng = new RNGCryptoServiceProvider())
    {
        rng.GetBytes(salt);
    }

    using (var SHA = new SHA256Managed())
    {
        byte[] saltPlusKey = salt.Concat(Encoding.UTF8.GetBytes(password)).ToArray();

        byte[] derivedKey = SHA.ComputeHash(saltPlusKey);

        for (int i = 0; i < iterations; i++)
        {
            derivedKey = SHA.ComputeHash(derivedKey);
        }

        return iterations.ToString() + ":" +
            Convert.ToBase64String(salt) + ":" +
            Convert.ToBase64String(derivedKey);
    }
}
```

The verification function works in typical fashion:

```
private bool VerifyStretchedPasswordSHA256(string hash, string password)
{
    string[] parts = hash.Split(':');

    int iterations = Convert.ToInt32(parts[0]);

    byte[] salt = Convert.FromBase64String(parts[1]);

    using (var SHA = new SHA256Managed())
    {
        byte[] saltPlusKey = salt.Concat(Encoding.UTF8.GetBytes(password)).ToArray();

        byte[] derivedKey = SHA.ComputeHash(saltPlusKey);

        for (int i = 0; i < iterations; i++)
        {
            derivedKey = SHA.ComputeHash(derivedKey);
        }
        return (parts[2] == Convert.ToBase64String(derivedKey));
    }
}
```

Here is a quick example of the methods being used:

```
string hash = StretchPasswordSHA256("password", 1000);

bool ok = VerifyStretchedPasswordSHA(hash,"password");
```

Salt Storage Alternatives

Throughout this chapter and others we have discussed salt values being stored with the hash values. This is by far the most common method of storing the salt. It's also the simplest.

Separate Salt Store

A separate salt store is where the salts are not stored with the hash, and instead are stored in a database or file. Layered security or granular access control is the primary argument for this. Proponents of this model like the fact that by enforcing access to the salt store it essentially makes the salt act as a key. This can provide the added security of a private key with the secure one-way nature of a hash function.

However, separate salt stores can bring more complexity to a solution. As you already know, complexity is our enemy. This complexity stems from maintaining the separate store. Be it a database or a file, access to this store must be controlled and secured. Is the access control strong enough? Are the salts backed up? Are the backups secured? Is the method of access/transport secured? As you can see, you should be ready for another aspect of the system that needs security.

On-the-Fly Salting

Some solutions have used input data as the salt and handled the stretching on the fly, opting not to store the salt. This is fine if the salt can be correctly reproduced at a later date by the user or application. However, this often opens the door to the use of low-entropy (insecure) salts.

Chapter Summary

- Password-Based Key Derivation Functions (PBKDF) are used to derive a cryptographically secure key from a low-entropy password. Like hash algorithms, PBKDFs are deterministic, meaning they will derive the same key given the same input, every time.
- Rfc2898DeriveBytes (PBKDF2) is the recommended key derivation method in .NET.
- A salt is a random piece of data used to increase the entropy of data derived from hash functions. Salts should be random and at least the same length of the hash: a 256-bit hash should use a 256-bit salt. Salts must be stored in a manner that allows them to be retrieved during the verification process. Salts are usually stored with the hash or derived key and do not *need* to be kept secret.
- Salts can also be stored separately from the data, but this usually adds an unnecessary increase to the complexity of the solution.
- Under best practices, a stretching function should take between 200 and 1000 milliseconds to execute.

- Stretching functions can be *iteration-centric* or *time-centric*. *Iteration-centric* functions execute for a fixed number of iterations but the overall execution time will vary depending on processor speed. *Time-centric* functions execute for a fixed amount of time regardless of processor speed; the faster the machine, the more iterations that will be completed in that timeframe. *Time-centric* functions can maintain a best-practice stretching time even if the application executes on faster processors.

Chapter Questions and Exercises

1. Explain the added protection provided through salting and stretching.
2. What is the recommended PBKDF to use in .NET?
3. Explain the different methods for storing salts.
4. Understand how to change iterations in a stretching function without making a breaking change to existing derived keys or passwords.
5. Explain how *time* can change while *iterations* stay the same.
6. Understand the concept of *time-centric* stretching and the problem that it solves.

Scenarios

1. You have a simple mobile application that encrypts data that is stored on the device. You don't want to store a symmetric key on the device because it could be compromised by an attacker with physical access. How could you use a PBKDF to solve your problem? What sacrifices might have to be made in terms of convenience? What are potential problems?

6 Symmetric Encryption

Symmetric Encryption: *A cryptographic process that uses the same secret key for its operation and, if applicable, for reversing the effects of the operation.*

Chapter Objectives

1. Understand the symmetric encryption model and the algorithms implemented in .NET.
2. Learn the functionality of the **SymmetricAlgorithm** base class and how encryption and decryption are handled.
3. Understand the purpose of padding.
4. Learn the different block-cipher modes in .NET and their security implications.
5. Know how to generate IVs and attach them to a ciphertext.
6. Learn which .NET algorithms can provide 128-bit security.

Symmetric algorithms are used to ensure the privacy of a particular piece of data. We use symmetric algorithms because they offer security, simplicity, and speed. Symmetric encryption requires an instance of a symmetric algorithm (access to the algorithm itself), a secret key, and the message to be encrypted. .NET offers a stable and practical model for symmetric encryption with access to secure algorithms and modes of operation. In this chapter we will cover different algorithms used for symmetric encryption, their relative strengths, and how they can be used in .NET.

Quick Start Example: AES256

This chapter covers many principles and different aspects of programming symmetric cryptography in .NET. Before we get into this we'll show you a quick encrypt and decrypt example, which will help provide some context to the principles in the chapter and give a starting point to those of you who are eager.

Our first method encrypts a variable-length byte array using a 32-byte key with the AES256 algorithm and returns the encrypted data:

```
byte[] Encrypt(byte[] data, byte[] key)
```

```
{
    using (AesManaged aes = new AesManaged())
    {
        aes.Key = key;

        using (ICryptoTransform encryptor = aes.CreateEncryptor())
        {
            byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);

            byte[] ciphertext = aes.IV.Concat(encryptedData).ToArray();

            aes.Clear();

            return ciphertext;
        }
    }
}
```

The next method will decrypt data presumably encrypted with the above method using the same key:

```
byte[] Decrypt(byte[] data, byte[] key)
{
    using (AesManaged aes = new AesManaged())
    {
        aes.IV = data.Take(aes.IV.Length).ToArray();
        aes.Key = key;

        using (ICryptoTransform decryptor = aes.CreateDecryptor())
        {
            byte[] plaintext = decryptor.TransformFinalBlock(data, aes.IV.Length, data.Leng
th - aes.IV.Length);

            aes.Clear();

            return plaintext;
        }
    }
}
```

In the above methods the **KeySize**, **Mode**, and **Padding** properties are defaulted to 256-bit, CBC mode, and PKCS7, respectively.

Block Ciphers in .NET

We explained in the overview that block ciphers encrypt data in blocks. This means that a 128-bit block cipher will take 128 bits of plaintext data and encrypt it to produce 128 bits of ciphertext (encrypted data). In modern secure applications, 128-bit (16-byte) block ciphers provide strong security and are more resistant to attacks than the previous generation of block ciphers, most of which were 64-bit. Currently, .NET only supports block ciphers; popular stream ciphers are not supported. However, some third-party libraries offer stream cipher implementations for .NET.

Advanced Encryption Standard (AES) and Rijndael

The current industry standard for block cipher algorithms is Advanced Encryption Standard (AES). AES is a 128-bit block cipher that can use 128, 192, or 256-bit keys. However, AES is only the name of the standard, not the algorithm. The algorithm that AES uses is called Rijndael. The Rijndael algorithm can support both block *and* key sizes of 128, 192, and 256 bits, but AES only implements Rijndael with a 128-bit block size. AES256 (AES with 256-bit keys) is the U.S government standard for data classified as Top Secret.

DES

Before AES, there was DES—the Data Encryption Standard. DES uses 64-bit blocks and 56-bit keys (**the .NET implementations use 64-bits, but only 56 bits provide data protection**). DES is extremely insecure and should never be used to protect data.

TripleDES

Once the security of DES started to unravel, a quick solution was needed and TripleDES (3DES) was the answer. The concept of TripleDES is exactly what it sounds like: DES x 3. TripleDES is a block cipher that uses three DES functions in sequence to product a 64-bit block. The increased security in TripleDES mainly stems from the longer key size than DES. Still, TripleDES is not secure for use in modern production environments and should only be used to integrate with legacy systems.

RC2

RC2 is a symmetric algorithm developed by Ron Rivest to be used in place of DES. It offers poor security compared to AES or Rijndael and is not recommended for use in modern secure systems.

Symmetric Algorithm Quick Comparison for .NET

Table 16 provides a quick comparison of the key and block sizes of the symmetric algorithms in .NET and whether they are considered secure for use in modern cryptosystems.

Table 16: Quick comparison of .NET symmetric encryption algorithm

Algorithm Name	Class Name	Available Key Lengths in .NET (bits)	Block Sizes in .NET (bits)	Recommended for Secure Production Environments (as of 2016)
AES (Rijndael)	Aes, AesManaged, AesCryptoServiceProvider	128, 192, 256	128	*Yes*
Rijndael	Rijndael, RijndaelManaged, RijndaelManagedTransform	128, 192, 256	128, 192, 256	Yes
TripleDES	TripleDES, TripleDESCryptoServiceProvider	64, 128, 192	64	No
DES	DES, DESCryptoServiceProvider	64	64	No
RC2	RC2, RC2CryptoServiceProvider	40-128	64	No

Block Cipher Modes

Block ciphers operate on blocks of data. But most data that needs to be encrypted is larger than an algorithm's block size. To determine how data spanning the length of multiple blocks will be processed, block cipher modes are used. For this reason, most developers will never use a block cipher directly, and will use it in *a mode of operation* instead.

The block cipher mode will determine how data is actually delivered to and processed by the block cipher algorithm. Implementations of symmetric algorithms in .NET default to Cipher Block Chaining (CBC) mode, a secure and robust cipher mode (its details are covered shortly).

Algorithms in .NET only allow the use of certain cipher modes. These vary on an algorithm-by-algorithm basis. Other modes may become available with later versions of .NET, or in third-party libraries. However, developers should be careful when implementing third-party libraries and modes that they are unfamiliar with operating. Some modes have specific security issues and rigid constraints for implementation. Diligent research is always best before implementing new modes.

The **CipherMode** enum in the **System.Security.Cryptography** namespace contains five block cipher modes. Algorithms that implement **SymmetricAlgorithm** can get or set their **CipherMode** through the **SymmetricAlgorithm.Mode** property.

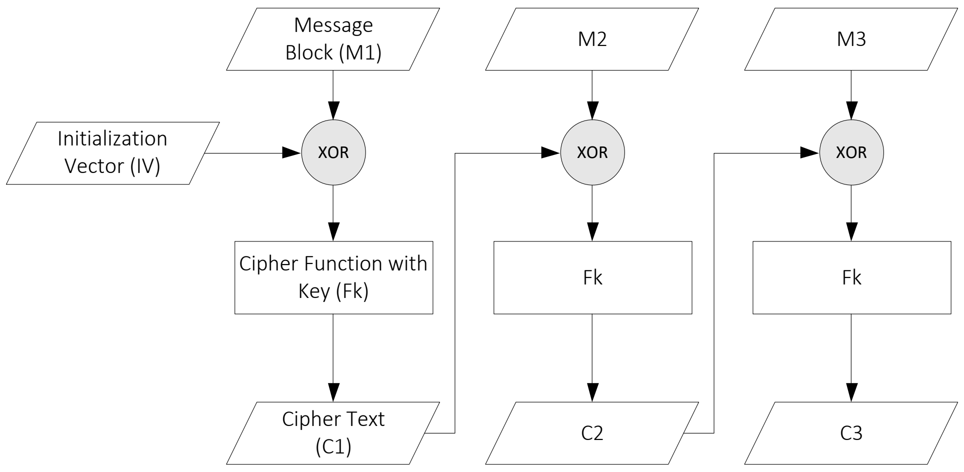
CipherMode enum elements:

- CBC
- CFB
- CTS
- ECB
- OFB

CBC Mode

The Cipher Block Chaining (CBC) mode introduces feedback (additional data) back into the encryption process. The initialization vector (IV) is XORed with the first message block before the first ciphertext block is produced. Before each plaintext block is encrypted, it is XORed with the previous cipher block. This process is illustrated in Figure 18. Even if a message contains identical blocks, the ciphertext output will not be the same. Due to this chaining behavior, if any part of the message is not intact, or incorrect, decryption will render a mangled ciphertext.

Figure 18: CBC Mode



A common variation of CBC mode, known as *chained CBC* (also known as *stateful CBC*), is able to reduce the requisite number of IVs for a series of messages. In regular (non-chained) CBC, each message generates a new IV, which after being used to encrypt the data is prepended to the message. However, this means that each message, or packet, is going to be larger due to the IV. Very high traffic systems could notice a difference where messages incur this additional overhead (unless you're Facebook or Google, I don't expect this will actually be the case). Chained CBC has been used to reduce the overhead that IVs create. In chained CBC the last message block being sent from a particular party is used as the IV for the next whole message.

Chained CBC introduces both logistical and security problems. In order for chained CBC to actually work there must be an underlying management system that maintains the *state* of the series of messages. Regular CBC mode works in a way that if one part of the message is corrupted or decrypts incorrectly the remainder of the message will not process correctly because incorrect data is being reintroduced throughout the rest of the message. Chained CBC follows this same pattern, but in a much larger scope, which makes it extremely vulnerable to processing issues. I don't recommend using chained CBC implementations.

CFB Mode

The Cipher Feedback (CFB) mode processes a plaintext incrementally using a shift register, rather than processing an entire block each time. Similar to how one bad bit will get mixed through an entire message using CBC mode, a bad bit in CFB mode can span multiple cycles of the shift register and have an effect on multiple blocks.

The size of the feedback (the number of bits that are used) can be changed using the **System.Security.Cryptography.SymmetricAlgorithm.FeedbackSize** property.

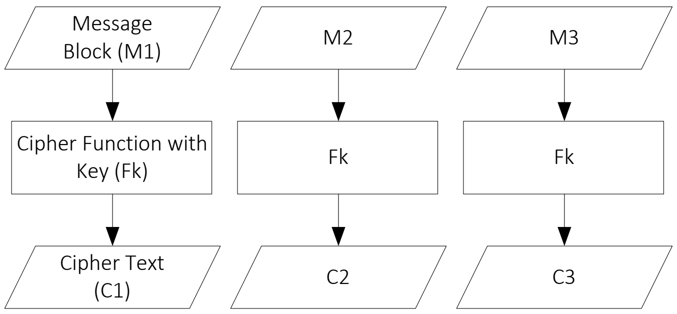
CTS Mode

The Cipher Text Stealing (CTS) mode can accept any length of plaintext and produce a ciphertext with the same length. CTS and CBC essentially work the same besides the last two blocks.

ECB Mode

The Electronic Codebook (ECB) mode processes each message block separately. This means that any blocks of plaintext that are identical and are in the same message, or in a different message encrypted with the same key, will be transformed into identical ciphertext blocks. If the plaintext to be encrypted contains substantial repetition, it is feasible for the ciphertext to be broken one block at a time. Also, it is possible for an active adversary to substitute and exchange individual blocks without detection. If a single bit of the ciphertext block is mangled, the entire corresponding plaintext block will also be mangled. To clarify, when a block is mangled in ECB, it will not interfere with other blocks in the message, as it would with a mode like CBC.

Figure 19: ECB Mode



OFB Mode

The Output Feedback (OFB) is very similar to CFB and processes pieces of a message rather than breaking it up into blocks and processing a block at a time (these modes do differ in terms of how the shift register is filled). Bad input bits will affect output on a one-to-one basis. Missing input bits can, however, corrupt the remainder of a message.

Which Modes are used with Which Algorithms?

At the time of this writing, every block cipher implementation in .NET uses at least ECB and CBC modes. Three algorithms in .NET can also use CFB mode. Table 17 lists the available cipher modes for each block cipher implemented in .NET through the **System.Security.Cryptography** namespace.

Table 17: Cipher mode availability for .NET algorithms

Algorithm in .NET	Available Cipher Modes
AES	ECB, CBC
DES	ECB, CBC, CFB
Rijndael	ECB, CBC
RC2	ECB, CBC, CFB
TripleDES	ECB, CBC, CFB

Adjusting the CipherMode

Cipher modes can be changed by setting the **Mode** property of a **SymmetricAlgorithm** instance to a different **CipherMode**. CBC mode is used by default in .NET and shouldn't be changed unless there is a specific and well-thought-out reason. Changing the cipher mode in a production system will result in a breaking change if there is data encrypted under a different cipher mode. For instance, persisted data that was encrypted with ECB will not decrypt correctly if the cipher mode is changed to CBC.

Below we set the **Mode** property on a new instance of **AesManaged**:

```
AesManaged aes = new AesManaged();
aes.Mode = CipherMode.CBC;
```

It should be noted that the cipher mode could be attached to the ciphertext similarly to how salts are often attached to hashes, allowing the cipher mode to be retrieved and set prior to decryption.

Security Issues Associated with ECB

Electronic Code Book (ECB) Mode is the simplest mode of operation for a block cipher because it creates a block of ciphertext for each block of plaintext. The threat here is that an attacker could actually compile a lookup of ciphertext blocks that are derived from a specific key and plaintext. Messages often contain identical data in identical positions, such as headers or boilerplate greetings. Attackers that know this can compile ciphertext blocks, derived from the same data, in an attempt to compromise a key. ECB mode should never be used in a secure production environment (*ever*).

Integrity is another concern with ECB mode. A ciphertext block could be switched in transit by an attacker without affecting the rest of the message. Even where the attacker doesn't know the key, he or she could insert data previously encrypted by the same key into a captured ciphertext, retransmit, and the recipient would be none the wiser.

CBC has traditionally been the go-to mode for eliminating these issues. However, other modes that introduce random data, such as a nonce, IVs, and feedback, can also provide additional protection. At the time of this writing, given what is available in .NET, CBC is the recommended mode.

Selecting a Cipher Mode

We discussed the problems with ECB last section and why CBC is the preferred mode to use in .NET. Another reason to use CBC mode is that it's very robust. We recommend using a random IV for CBC. .NET implementations of SymmetricAlgorithm will generate a random IV, which is obviously easier for the programmer and therefore makes for less error-prone solutions. Of course, you can also randomly generate your own IV if you want, but don't forget to use the RNGCryptoServiceProvider instead of Random (this will be shown in the upcoming IV section).

Initializations Vectors (IVs)

Initialization Vectors (IVs) refer to data that is used to add randomness to the first part of a cryptographic function to help break up any patterns in the ciphertext that could be visible to an attacker. This unpredictable data is the backbone of the security of many cipher modes (CBC is a chief example). It's no surprise that IVs are commonly implemented incorrectly by developers and result in security weakness. Random IVs are recommended for their security and simplicity in generation (compared to counter or nonce IVs, where a secure management system is necessary).

Security Issues Associated with IV Reuse

The security of CBC mode is only partially reduced with IV reuse compared to that of some nonce-dependent modes and stream ciphers where nonce or IV reuse can cripple security. Fixed IVs (a hardcoded IV that doesn't change between messages) should never be used.

Reuse can be fairly predictable with random IVs. For instance, we can expect a collision in 128-bit random IVs in about 2^{64} IV generations. However, this is only a concern where the same key is being used.

Padding Modes

When the length of a plaintext is not a perfect multiple of the block cipher's block size, and the cipher mode requires this, padding is used to fill the void. Padding must be attached in a way so that it can be identified and properly removed in the decryption process in a way that does not damage the original plaintext. A padding mode does not carry the security ramifications that cipher modes do; reversibility is what matters most with padding.

The **PaddingMode** enum in the **System.Security.Cryptography** namespace provides an easy way to handle padding. Table 18 describes each of the five modes within the **PaddingMode** enum.

Table 18: Modes in the PaddingMode Enum

PaddingMode	Description
ANSIX923	The ANSIX923 padding string consists of a sequence of bytes filled with zeros before the length.
ISO10126	The ISO10126 padding string consists of random data before the length.
None	No padding is performed.
PKCS7	The PKCS #7 padding string consists of a sequence of bytes, each of which is equal to the total number of padding bytes added.
Zeroes	The padding string consists of bytes set to zero.

PKCS7 is also the default padding mode used in .NET symmetric encryption algorithms. The **Padding** property can be set like this in any **SymmetricAlgorithm** subclass:

```
AesManaged aes = new AesManaged()
aes.Padding = PaddingMode.PKCS7;
```

***Invalid padding**: Invalid padding should be treated the same as an authentication failure. By default, the .NET block cipher implementations will throw an error if the specified padding is not formatted correctly.*

Keys

The security that a symmetric algorithm can provide is dependent on the strength of the key being used. Symmetric algorithms have much looser requirements for key material than asymmetric algorithms. Asymmetric (public-key) algorithms must have key material generated for a particular algorithm because the public and private keys are mathematically related. Subsequently, mathematical differences in public-key algorithms prevent keys from being used between algorithms. For instance, we cannot use an RSA key in an ECDSA algorithm. Nor can we simply derive these keys from a PBKDF, HMAC, or hash algorithm like we can for symmetric keys where key length is the deciding factor for whether a key will work in any given symmetric algorithm. Because length is the main factor in classifying a symmetric key, these keys are usually referred to by their key size: 64-bit, 128-bit, 192-bit, 256-bit, and so on. A key for one 128-bit block cipher could work for another. Along these same lines, we could truncate a 256-bit AES key down to a smaller length to work in a different algorithm. However, this type of process (truncating a key to fit another key size) is usually unnecessary because symmetric key material is easily derived.

Programming Symmetric Algorithms

The Symmetric Algorithm Base Class

All block ciphers in .NET must implement the **SymmetricAlgorithm** abstract class. This class has a simple and intuitive interface that helps when writing generic methods and classes. Table 19 (page 85) and Table 20 (page 86) contain the properties and methods of the **SymmetricAlgorithm** base class.

Creating an Instance

An instance of a block cipher can be created directly using the new keyword. This is preferred because it is explicit:

```
RijndaelManaged rijnManaged = new RijndaelManaged();
```

The factory-style **Create** method can be used to create an instance of a .NET block cipher by supplying the algorithm name as an argument. Below, an instance of **AesCryptoServiceProvider** with 256-bit keys (AES256) is created:

```
SymmetricAlgorithm aes = SymmetricAlgorithm.Create("Aes");
```

Supplying zero arguments to **Create** in the current version of .NET (4.6) will create an instance of **RijndaelManaged** with 256-bit keys:

```
SymmetricAlgorithm rijndael = SymmetricAlgorithm.Create();
```

Setting the Key

A random key will be set by default, but this is of little help when the developer needs to use a predefined key. The **Key** property can be used to get or set the private byte array key used by the algorithm:

```
byte[] key = ...

using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
{
    rijndaelManaged.Key = key;
```

What about the IV?

IVs are accessed through the **IV** property in **SymmetricAlgorithm** subclasses and will be randomly generated by default. So, you will only need to set it if you are implementing a custom IV. Below we use **RNGCryptoServiceProvider** to set the IV of an **AesManaged** instance (but as we’ve stated this is unnecessary because it’s already securely generated during instantiation):

```
using(AesManaged aes = new AesManaged())
{
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(aes.IV);
```

Standard implementation of CBC mode will require the IV to be transmitted with the ciphertext (usually prepended). This will be covered in upcoming examples.

SymmetricAlgorithm Methods

Table 19: Methods of the SymmetricAlgorithm class

Method Name	Description
Clear()	Releases all resources used by the class.
Create(string)	Creates the specified cryptographic object used to perform the symmetric algorithm.
Create()	Creates a default cryptographic object used to perform the symmetric algorithm.
CreateDecryptor(byte[],byte[])	When overridden in a derived class, creates a symmetric decryptor object with the specified Key property and initialization vector (IV).
CreateDecryptor()	Creates a symmetric decryptor object with the current Key property and initialization vector (IV).
CreateEncryptor(byte[]byte[])	When overridden in a derived class, creates a symmetric encryptor object with the specified Key property and initialization vector (IV).
CreateEncryptor()	Creates a symmetric encryptor object with the current Key property and initialization vector (IV).
Dispose(bool)	Releases the unmanaged resources used by the SymmetricAlgorithm and optionally releases the managed resources.
Dispose()	Releases all resources used by the current instance of the class.
GenerateIV()	When overridden in a derived class, generates a random initialization vector (IV) to use for the algorithm.
GenerateKey()	When overridden in a derived class, generates a random key to use for the algorithm.
SymmetricAlgortithm()	Initializes a new instance of the SymmetricAlgorithm class.
ValidKeySize(int)	Determines whether the specified key size is valid for the current algorithm.

SymmetricAlgorithm Properties

Table 20: Properties of the SymmetricAlgorithm class

Property Name	Description
BlockSize	Gets or sets the block size, in bits, of the cryptographic operation.
FeedbackSize	Gets or sets the feedback size, in bits, of the cryptographic operation.
IV	Gets or sets the initialization vector (IV) for the symmetric algorithm.
Key	Gets or sets the secret key for the symmetric algorithm.
KeySize	Gets or sets the size, in bits, of the secret key used by the symmetric algorithm.
LegalBlockSizes	Gets the block sizes, in bits, that are supported by the symmetric algorithm.
LegalKeySizes	Gets the key sizes, in bits, that are supported by the symmetric algorithm.
Mode	Gets or sets the mode for operation of the symmetric algorithm.
Padding	Gets or sets the padding mode used in the symmetric algorithm.

Default Block Cipher Configuration: *By default, .NET block ciphers are instantiated using a random IV, a random key, CBC mode, and PKCS7 padding. For algorithms with multiple key sizes, .NET will default to the largest size. This gives developers a robust and secure implementation of the algorithm by default.*

The CryptoStream Class

CryptoStream is a class implemented by the CLR that allows reading and writing of data to a stream intended specifically for cryptographic purposes. Being that this is stream data you will not have to worry about the data in blocks; **CryptoStream** will interact with the **ICryptoTransform** object to actually perform the block-based cryptographic operation such as encryption or decryption. Data streamed in will be encrypted and data streamed out will be decrypted.

What is an ICryptoTransform? *The ICryptoTransform interface defines the basic operations of cryptographic transformations. This is commonly implemented in objects that perform cryptographic operations on byte-array data in memory or in a Stream.*

Since numerous cryptographic operations often make up a single cryptographic function, **CryptoStreams** can be used to consolidate the output of these individual operations rather than having to manage each one independently and combine their results later. An example of this could be a method written by a developer to encrypt and authenticate data. Such a method will incorporate many steps, each of which involving different classes and objects that have to process data. Here, rather than storing and managing data that all of these objects return, they could write to a **CryptoStream** instead. Objects implementing the **ICryptoTransform** interface can hook up directly to a **CryptoStream**. **CryptoStreams** are used in conjunction with another **Stream** object such as **MemoryStream**, an **ICryptoTransform** object, and a **CryptoStreamMode** to indicate whether the operation will be performing I/O as a read (**CryptoStreamMode.Read**) or a write (**CryptoStreamMode.Write**). **CryptoStream** objects can also be chained with other stream objects (an example will be covered later in the chapter).

Below is sample code using **AesManaged** that relies on the relationship between **ICryptoTransform**, **MemoryStream**, and **CryptoStream** objects. It should be noted that **CryptoStream** objects must be closed using the **Close** method before data can be accessed by the underlying stream such as **MemoryStream**.

```
using (var BlockCipher = new AesManaged())
{
    using (ICryptoTransform encryptor = BlockCipher.CreateEncryptor())

    using (MemoryStream memStrm = new MemoryStream())
    {
        CryptoStream crptStrm = new CryptoStream(memStrm, encryptor,
CryptoStreamMode.Write);

        memStrm.Write(BlockCipher.IV, 0, BlockCipher.IV.Length);

        crptStrm.Write(data, 0, data.Length);
        crptStrm.FlushFinalBlock();
        crptStrm.Close();

        BlockCipher.Clear();

        ciphertext = memStrm.ToArray();
    }
}
```

Using a **CryptoStream** also introduces the possibility of using other helpful objects, such as the **ToBase64Transform** or **FromBase64Transform** classes, which convert **CryptoStream** instances to and from Base64.

TransformFinalBlock

Where you can afford to handle complete data in memory (this is most cases besides large files or streams), the **TransformFinalBlock** method of the **ICryptoTransform** interface makes performing cryptographic operations simpler and less verbose than using a **CryptoStream**. The same code form the previous **CryptoStream** example is shown below and refactored to use the **TransformFinalBlock** method. It benefits from about a 50% reduction in overall code length, plus it's just simpler. Most of the symmetric encryption examples in this book will use **TransformFinalBlock**. The code below performs symmetric encryption using **TransformFinalBlock** directly:

```
using (var blockCipher = new AesManaged())
{
    using (ICryptoTransform encryptor = blockCipher.CreateEncryptor())
    {
        byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);

        return blockCipher.IV.Concat(encryptedData).ToArray();
    }
}
```

Releasing Resources

An instance of the symmetric algorithm class will hold resources while it is being used. These are things like keys, data, and of course, the memory allocated to the instance itself. Other objects used in conjunction with the cipher will also hold resources and need to be managed. Releasing resources in cryptographic code is important for performance and security. The scope and lifetime of objects containing secret data should always be carefully managed and kept to a minimum.

IDisposable

Classes that implement the **IDisposable** interface can take advantage of the **using** keyword to implicitly release their allocated resources. **ICryptoTransform**, **MemoryStream**, and **CryptoStream**, are all used closely with **SymmetricAlgorithm** implementations and all implement **IDisposable**. Because of how .NET garbage collection works, symmetric algorithm objects containing keys and plaintexts will not be zeroed or overwritten; the memory will only be freed. **Dispose** should be called to manually clear this sensitive memory.

Steps for Simple Encryption and Decryption

When people are new to encryption in .NET it can be difficult to know which objects to use, how to use them, and in what order things need to happen. The same objects will be used in both encryption and decryption. Some objects will need to be placed in different modes. These processes can be broken down to five basic steps (these do not include processing **Stream** data. For this, refer to other examples that use **CryptoStream** objects.)

Encryption

1. Create an instance of the symmetric encryption algorithm like **AES** or **Rijndael**.
2. Set the algorithm's **Key** property with a symmetric encryption key in byte-array format.
3. Create an encryptor object using the algorithm's **CreateEncryptor** method.
4. Encrypt the data using the **TransformFinalBlock** method of the encryptor object. This method will take the plaintext data as a parameter.
5. If CBC mode is being used (it is by default), attach the algorithm's **IV** property to the encrypted data from step 4. This is commonly achieved by concatenating the IV byte array with the encrypted data byte array.

Decryption

1. Create an instance of the symmetric encryption algorithm like **AES** or **Rijndael**.
2. Set the algorithm's **Key** property with a symmetric encryption key in byte-array format.
3. If CBC mode is being used, remove the IV from the encrypted data and set the algorithm's **IV** property.
4. Create a decryptor object using the algorithm's **CreateDecryptor** method.
5. Decrypt the data using the **TransformFinalBlock** method of the decryptor object. This method will take the encrypted data as a parameter.

Example: Simple AES Encrypt and Decrypt

A simple interface for a symmetric encryption method will take the plaintext data that needs to be encrypted and a key as arguments, and return a ciphertext. .NET's implementations of standard encryption algorithms use sane defaults with the most robust and secure modes available: CBC Mode, PKCS7 Padding Mode, random IV, random key, and largest key size. Creating an instance of an algorithm that defaults to these properties means developers don't have to worry about manually setting them (and possibly screwing them up). It also makes for shorter, cleaner code. In this example **TransformFinalBlock** will be used instead of **CryptoStream** and **MemoryStream.**

A few key points in the example should be noted:

1. Resources are disposed of with a **using** statement.
2. The algorithm's **Key** property is set from the method parameter key.
3. The algorithm's **IV** is attached to the ciphertext because it will need to be transmitted to the recipient for use in decryption.
4. The algorithm's **Clear** method is explicitly called after its work is done.

```
byte[] Encrypt(byte[] data, byte[] key)
{
    using (AesManaged aes = new AesManaged())
    {
        aes.Key = key;

        using (ICryptoTransform encryptor = aes.CreateEncryptor())
        {
            byte[] encryptedData = encryptor.TransformFinalBlock(data,0,data.Length);

            byte[] ciphertext = aes.IV.Concat(encryptedData).ToArray();

            aes.Clear();

            return ciphertext;
        }
    }
}
```

The logical progression of the decryption method is very similar to that of the encryption method. Key points in the decryption method are:

1. Resources are disposed of with a **using** statement.
2. The algorithm's **Key** property is set from the method parameter key.
3. The algorithm's **IV** property is set from the IV that is removed from the ciphertext (data).
4. The **TransformFinalBlock** method takes into account the IV length when operating on the data.
5. The algorithm's **Clear** method is explicitly called after its work is done.

```
byte[] Decrypt(byte[] data, byte[] key)
{
    using (AesManaged aes = new AesManaged())
    {
        aes.IV = data.Take(aes.IV.Length).ToArray();
        aes.Key = key;

        using (ICryptoTransform decryptor = aes.CreateDecryptor())
        {
            byte[] plaintext =decryptor.TransformFinalBlock(data, aes.IV.Length, 
data.Length - aes.IV.Length);

            aes.Clear();

            return plaintext;
        }
    }
}
```

Example: Encryption and Decryption with Generics

Favoring generic design is axiomatic in an object-oriented world. It improves software quality and productivity full-circle. Generic encryption and decryption methods can help the developer save time and avoid errors when building cryptographic applications. Generally, methods like this should expose a simple interface and allow the developer to supply an algorithm through the generic type parameter (T).

Correctness and robustness are important with these types of methods as they may be implemented in a variety of environments by a variety of people. The interface should be kept intuitive, simple, and hard to screw up. The biggest challenge is balancing functionality and usability with security. What kind of control do you give those who consume the interface? Do you allow them to change the **CipherMode** or **PaddingMode**? Do you offer text/string encoding options? For simplicity, our methods will have the following functionality and expected behavior:

1. A generic type parameter must implement the **SymmetricAlgorithm** base class and have a default (parameterless constructor).
2. *Encrypt<T>* and *Decrypt<T>* methods that accept byte-array data and return byte-array data.

```
public byte[] Encrypt<T>(byte[] data, byte[] key) where T : SymmetricAlgorithm, new()
{
    using (var algorithm = new T())
    {
        algorithm.Key = key;

        using (ICryptoTransform encryptor = algorithm.CreateEncryptor())
        {
            byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);
            byte[] ciphertext = algorithm.IV.Concat(encryptedData).ToArray();

            algorithm.Clear();

            return ciphertext;
        }
    }
}
```

```
public byte[] Decrypt<T>(byte[] data, byte[] key) where T : SymmetricAlgorithm, new()
{
    using (var algorithm = new T())
    {
        algorithm.IV = data.Take(algorithm.IV.Length).ToArray();
        algorithm.Key = key;

        using (ICryptoTransform decryptor = algorithm.CreateDecryptor())
        {
            byte[] plaintext = decryptor.TransformFinalBlock(data, algorithm.IV.Length, 
data.Length - algorithm.IV.Length);
```

```
            algorithm.Clear();

            return plaintext;
        }
    }
}
```

The code below shows how these methods are used with **AesManaged**:

```
byte[] dummyData = new byte[300];

byte[] unsafeKey = new byte[32];

byte[] ciphertext = Encrypt<AesManaged>(dummyData, unsafeKey);

byte[] plaintext = Decrypt<AesManaged>(ciphertext, unsafeKey);
```

It should be noted that the generic methods make the following assumptions.

1. Data validation has been performed as to the length and format of the input data.
2. Any error messages have been caught by the developer and were not allowed to trickle up to an attacker.

Example: Excluding Unsafe Algorithms

Consumers of the generic methods from last example have no control over the behavior of the underlying properties of the algorithm specified in the generic type parameter T (as long as they are not using a user defined type). In most cases this is the kind of locked-down interface developers want to expose from cryptographic code. Eliminating choices means eliminating mistakes that stem from bad choices such as using ECB mode or a fixed IV. However, consumers of the methods could still use a substandard algorithm like **DES**. Unless legacy applications are being supported by algorithms like **TripleDES**, strong algorithms such as **Aes** and **Rijndael** should be the only acceptable choices and at some point developers consuming the methods must be educated on this.

Disallowing algorithms that are not "whitelisted" is another option to restrict consumers of the generic methods from using insecure algorithms. This could be as simple as a type check on the generic type parameter (T) via a bool method the precipitates an exception or returns a null value in the encryption or decryption methods. However, it's up to the developer how this is handled. The code below shows a simple type whitelist method:

```
public bool IsSafeAlgorithm()
{
    Type algType = typeof(T);

    if (algType == typeof(AesManaged) ||
        algType == typeof(AesCryptoServiceProvider) ||
        algType == typeof(RijndaelManaged))
        return true;
    else
        return false;
}
```

This could be incorporated into the *Encrypt<T>* and *Decrypt<T>* methods like so:

```
public byte[] Encrypt<T>(byte[] data, byte[] key) where T : SymmetricAlgorithm, new()
{
    if (!IsSafeAlgorithm()) return null;
```

```
public byte[] Decrypt<T>(byte[] data, byte[] key) where T : SymmetricAlgorithm, new()
{
    if (!IsSafeAlgorithm()) return null;
```

Example: AES256 String Encryption

When you visit forums or message boards about encryption there always seems to be a demand for “string encryption examples.”

This example builds on the *Simple AES Encryption and Decryption* example from earlier. But instead of working exclusively with byte-array data, we’re going to work with string data. The plaintext input, key, and ciphertext output will all be strings. We are using UTF8 encoding to format strings before encryption and after decryption, and Base64 to handle ciphertext string data (post-encryption, pre-decryption). To add robustness to the solution, we are using a SHA256 hash of the input string key as the actual AES key. This allows strings of various lengths to be used as keys; otherwise a 32-byte string would have to be used every time (you could also use a PBKDF like those covered last chapter). As usual, we are using AES with 256-bit keys, CBC mode, and PKCS7 padding (the default configuration for AES in .NET at the time of this writing).

The first method is *Encrypt*. It takes data and a key as string input and returns a Base64 ciphertext:

```
string Encrypt(string stringData, string stringKey)
{
    byte[] data = Encoding.UTF8.GetBytes(stringData);
    byte[] key = Encoding.UTF8.GetBytes(stringKey);

    using(SHA256Managed sha256 = new SHA256Managed())
    using (AesManaged aes = new AesManaged())
    {
        aes.Key = sha256.ComputeHash(key);

        using (ICryptoTransform encryptor = aes.CreateEncryptor())
        {
            byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);

            byte[] ciphertext = aes.IV.Concat(encryptedData).ToArray();

            aes.Clear();

            return Convert.ToBase64String(ciphertext);
        }
    }
}
```

The next method is *Decrypt*. A Base64 ciphertext is expected as the string input along with the string key used for encryption. The UTF8-encoded plaintext string is returned:

```
string Decrypt(string stringData, string stringKey)
{
    byte[] data = Convert.FromBase64String(stringData);
    byte[] key = Encoding.UTF8.GetBytes(stringKey);

    using (SHA256Managed sha256 = new SHA256Managed())
    using (AesManaged aes = new AesManaged())
    {
        aes.IV = data.Take(aes.IV.Length).ToArray();
        aes.Key = sha256.ComputeHash(key);

        using (ICryptoTransform decryptor = aes.CreateDecryptor())
        {
            byte[] plaintext = decryptor.TransformFinalBlock(data, aes.IV.Length, 
data.Length - aes.IV.Length);

            aes.Clear();

            return Encoding.UTF8.GetString(plaintext);
        }
    }
}
```

The *Encrypt* and *Decrypt* methods can be used like so (remember, don't hard-code your key into your application code):

```
string data = "hello";
string key = "$LIWltnbkry3857KSUcRghg72h!"; //Dummy key.

string ciphertext = Encrypt(data, key);

string plaintext = Decrypt(ciphertext, key);
```

Example: Full Rijndael 256 (256-bit keys and blocks)

Until this point, we've only used AES256 with 128-bit blocks for our symmetric encryption examples. This uses the full Rijndael 256 algorithm where both the key and block sizes are 256 bits (recall that Rijndael can use 128-, 192- and 256-bit blocks and keys). The code below follows the same format as the simple AES encryption examples but sets the RijndaelManaged block size to 256. Note that this cannot be done using the AesManaged class, which will only work with 128-bit blocks. With the change in block size the IV length will automatically change to 256 bits (32 bytes):

```
byte[] Rijndael256Encrypt(byte[] data, byte[] key)
{
    using (RijndaelManaged rijn = new RijndaelManaged())
    {
        rijn.BlockSize = 256;
        rijn.Key = key;

        using (ICryptoTransform encryptor = rijn.CreateEncryptor())
        {
            byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);

            byte[] ciphertext = rijn.IV.Concat(encryptedData).ToArray();

            rijn.Clear();

            return ciphertext;
        }
    }
}
```

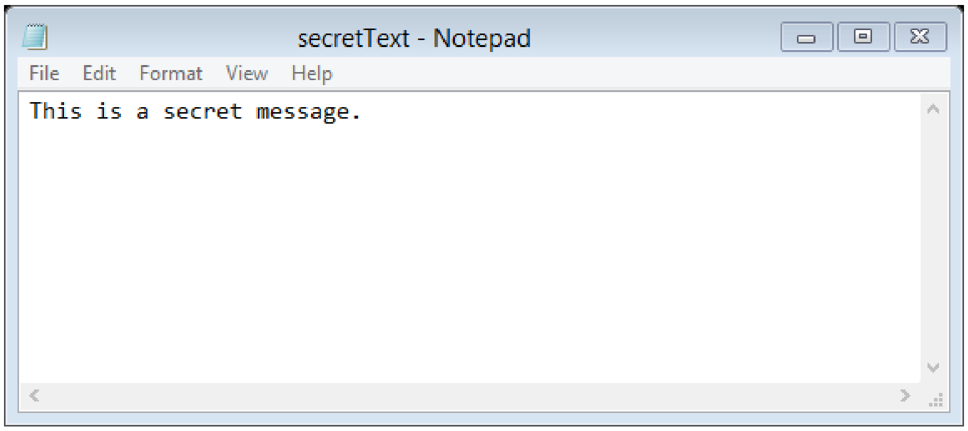
```
byte[] Rijndael256Decrypt(byte[] data, byte[] key)
{
    using (RijndaelManaged rijn = new RijndaelManaged())
    {
        rijn.BlockSize = 256;
        rijn.IV = data.Take(rijn.IV.Length).ToArray();
        rijn.Key = key;

        using (ICryptoTransform decryptor = rijn.CreateDecryptor())
        {
            byte[] plaintext = decryptor.TransformFinalBlock(data, rijn.IV.Length,
data.Length - rijn.IV.Length);

            rijn.Clear();

            return plaintext;
        }
    }
}
```

Example: Encrypt and Decrypt a Small Text File

This example builds on the last one: *AES256 String Encryption*. Here, we'll read text from a text file, encrypt it, and write it back to the file. We'll then decrypt it, returning it to the original plaintext. Base64 will be used to keep the ciphertext in a consistent text format, allowing it to be transmitted easily as well (binary could be used to save space using the **File.ReadAllBytes** and **File.WriteAllBytes** methods, but should not be used where the ciphertext needs to be transmitted or stored as text). Let's get started!

First, we'll create our text file, *secretText.txt*, with the message we want to protect. Figure 20 shows a short message in *secretText.txt*.

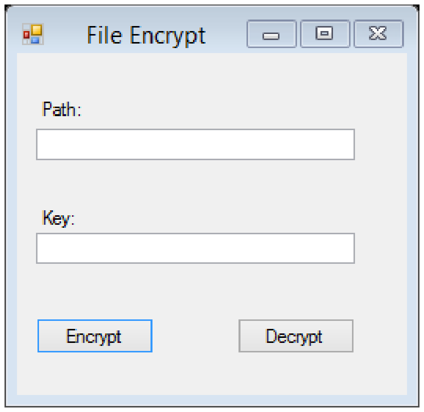
Figure 20: Plaintext message in .txt file



```
This is a secret message.
```

Next, we'll build a very simple Windows Forms application with textboxes for a file path and symmetric key, and buttons for encrypting and decrypting. Figure 21 illustrates an example form.

Figure 21: The Form



We'll write two methods in the Form code, *FileEncrypt* and *FileDecrypt* that specify the file path and key as parameters. Both will read the existing text from the text file using the **File.ReadAllText** method. This is then encrypted or decrypted (presumably using the *Encrypt* and *Decrypt* methods from the last example) and the result is written back to the file using **File.WriteAllText**:

```
void FileEncrypt(string path, string key)
{
    string text = File.ReadAllText(path);

    File.WriteAllText(path, Encrypt(text, key));
}
```

```
void FileDecrypt(string path, string key)
{
    string text = File.ReadAllText(path);

    File.WriteAllText(path, Decrypt(text, key));
}
```

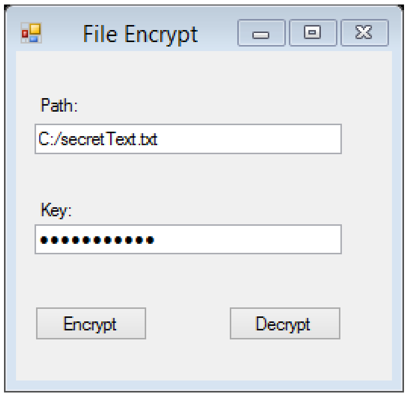
Our button click code for the *Encrypt* and *Decrypt* buttons will call the *FileEncrypt* and *FileDecrypt* methods with the textbox data (we separated these from the methods above to maintain better portability):

```
private void encryptButton_Click(object sender, EventArgs e)
{
    FileEncrypt(pathBox.Text, keyBox.Text);
}
```

```
private void decryptButton_Click(object sender, EventArgs e)
{
    FileDecrypt(pathBox.Text, keyBox.Text);
}
```

Figure 22 shows the form in use with an example file path and password.

Figure 22: Specifying the path and the key

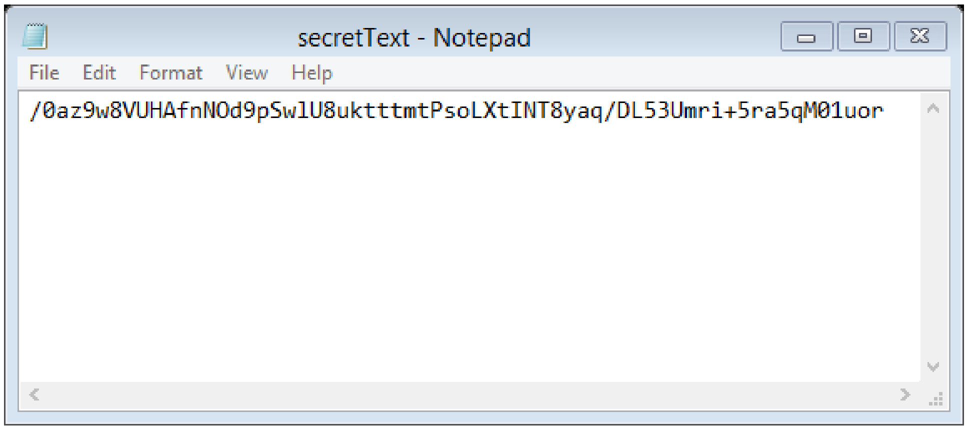


Path:
C:/secretText.txt

Key:
●●●●●●●●●●●

After hitting *Encrypt*, the plaintext in the file is replaced with the Base64 ciphertext. Figure 23 shows the encrypted contents of the text file (this will look different than your ciphertext).

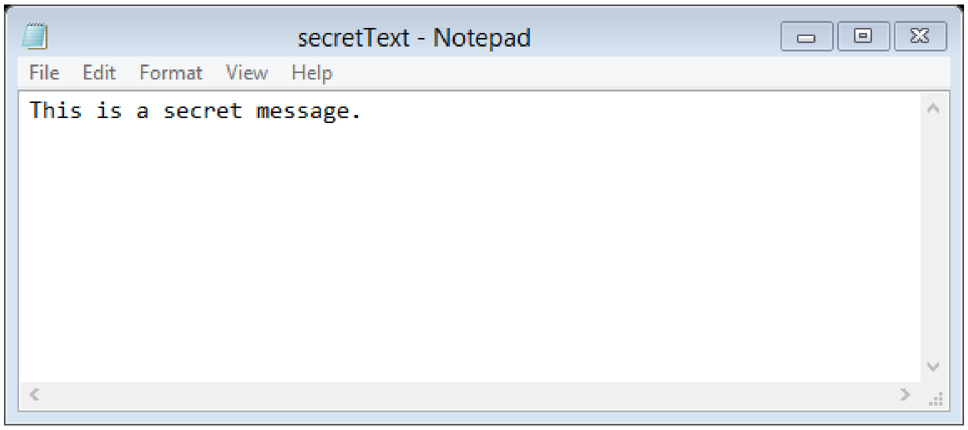
Figure 23: The encrypted message



```
/0az9w8VUHAfnNOd9pSwlU8ukttttmtPsoLXtINT8yaq/DL53Umri+5ra5qM01uor
```

Clicking *Decrypt* will display the decrypted (original) message, provided the password is correct. Figure 24 shows the original plaintext message after decryption.

Figure 24: The decrypted message



```
This is a secret message.
```

Remember, **Stream**/**CryptoStream** should be used for cryptographic operations on large files or resources.

Example: Encrypting to a File using Chained Streams

Here's a short example of using an **AesManaged** instance, a **FileStream**, a **CryptoStream**, and a **StreamWriter** to encrypt a plaintext message and write it to a bin file:

```
using (AesManaged aes = new AesManaged())
{
  using (ICryptoTransform encryptor = aes.CreateEncryptor())
  using (Stream fs = File.Create("secureFile.bin"))
  using (Stream cs = new CryptoStream(fs, encryptor, CryptoStreamMode.Write))
  using (StreamWriter sw = new StreamWriter(cs))
      sw.Write("This is a secret message");
}
```

One-Time Pads

A One-time pad (OTP) is a method of encrypting data where each bit of plaintext data is allotted a corresponding key bit. The same key is used to encrypt and decrypt, which technically characterizes the OTP as symmetric key encryption, although it is not in the block cipher or stream cipher family. OTPs offer the strongest level of data security but are impractical for most real scenarios due to the overhead of managing large unique keys. Some levels of secrecy justify the cost of OTPs; these might include espionage, national security matters, or important corporate data.

A 400-byte plaintext could be encrypted using an OTP, which would require a 400-byte key, and produce a 400-byte ciphertext. Keys are usually created using randomly generated material that matches the length of the plaintext. Many simple OTP implementations XOR the plaintext against the random key to get the ciphertext. A significant benefit here is that you can decrypt with the same XOR function.

It's easy to see that the biggest drawback from OTPs is managing keys that could easily be gigabytes in size (if the plaintext was this size). This adds to the key management issues associated with block ciphers and symmetric key systems in general. The fact that keys must also be unique means that where an entity can rely on a single symmetric key to encrypt many plaintexts (until a new key should be generated) in a symmetric system, a system relying on OTPs must generate a new one-time key for each plaintext. Those who have never worked with OTPs often underestimate how much of a key management issue this can present and should think carefully before implementing OTPs in a production environment. We recommend against it. Table 21 compares the storage overhead for key management between OTPs and AES256.

Table 21: Comparing OTP and AES256 Key Management Overhead

Cipher Method	Number of Messages at 1KB Each	Key Size	Minimum Number of Keys	Minimum Size of Key Storage
AES256	50000	256 bits	1	32 bytes
One-time pad (OTP)	50000	Size of message, here this is 1KB	50000	51200000 bytes

Example: OTP XOR

.NET does not offer any library implementations of OTPs. Writing an OTP encrypt and decrypt function, however, is straightforward and quite a bit easier than successfully writing a block cipher solution in .NET. Below, a simple OTP XOR function is written alongside a function to produce a key.

GetOneTimeKey takes the key length as an argument and returns a cryptographically random byte array to use as the key. Keep in mind, this key must be securely managed from the time of generation.

```
byte[] GetOneTimeKey(long size)
{
    byte[] key = new byte[size];

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(key);
    return key;
}
```

The actual encryption and decryption takes place in the *OtpXor* method. As previously discussed, the XOR functionality makes this one function capable of handling encryption and decryption.

```
byte[] OtpXor(byte[] plaintext, byte[] oneTimeKey)
{
    byte[] otp = new byte[plaintext.Length];

    for (int i = 0; i < plaintext.Length; i++)
    {
        otp[i] = (byte)((int)plaintext[i] ^ (int)oneTimeKey[i]);
    }

    return otp;
}
```

Calling *GetOneTimeKey* to generate a key and *OtpXor* is straightforward, but the developer will need to make sure that key management is in place to safely handle and store the key. Below, a quick example uses *OtpXor* to encrypt and decrypt a string.

```
byte[] data = Encoding.UTF8.GetBytes("Hello I'm some data");

byte[] oneTimeKey = GetOneTimeKey(data.Length);

byte[] cipherText = OtpXor(data,oneTimeKey);

string originalPlaintext = Encoding.UTF8.GetString(OtpXor(cipherText, oneTimeKey));
```

OTP Implementation Tips

Theoretically speaking, the one-time pad is the only unbreakable type of cipher, provided it is securely implemented. This, however, can prove to be a challenge. Here are few helpful guidelines, most of which involve secure key management:

1. Keys must be at least as long as the plaintext.
2. Keys can never be reused.
3. Keys must be highly random and generated from an unpredictable source.

4. Key storage media must be secure. A key's security can only be considered as strong as what is protecting it. If an OTP key is stored on a Windows machine that an attacker could compromise in 2^{64} attempts, the OTPs now only carry a 64-bit security level. This is an important issue.
5. Handle sensitive data for the shortest amount of time possible, know what comes into contact with the sensitive data (know its scope), and wipe state when finished.

Testing and Configuring

Bad things happen to good people. Analogously, good developers sometimes make bad mistakes. Whether these mistakes make their way into a production environment is often a result of sound testing and code review.

Configuring for Security and Robustness

.NET does a good job of putting together good default implementations for algorithms. Nevertheless, verify that your cryptographic solution is configured for security and robustness:

- Aes or Rijndael
- 256-bit keys (if possible) that are highly random
- Random IV
- CBC Mode
- PKCS7 Padding

Keys

The biggest mistake made when writing symmetric encryption code is improper handling and storage of keys. The problem with key management is its scope. Keys can (and need to) traverse many physical and logical boundaries, and have to deal with issues outside of just writing cryptographic code.

Example: Accidental Disclosure of Secrets

Cryptographic code needs extra scrutiny to ensure that secrets are maintained in the appropriate scope. It's not hard to accidentally return the plaintext from a function instead of the ciphertext, or attach a key to a message instead of an HMAC or an IV.

These types of errors can take place on a single line of code:

```
public byte[] BADEncrypt(byte[] data, byte[] key)
{
    using (var algorithm = new AesManaged())
    {
        algorithm.KeySize = 128;
        algorithm.Key = key;
        algorithm.IV = key;

        using (ICryptoTransform encryptor = algorithm.CreateEncryptor())
        {
            byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);
            byte[] ciphertext = algorithm.IV.Concat(encryptedData).ToArray();

            algorithm.Clear();

            return ciphertext;
        }
    }
}
```

The problem is that the IV is set with the key value. If the decryption method makes the same mistake it will even further exacerbate the issue because the methods will appear to work correctly and the error will go undiscovered until “The Big Data Breach” happens. But how likely is it that the decryption method would also make this same mistake? Considering that most developers will copy their encryption function, paste it as the decryption function, and simply switch around a few lines of code to make it work, it’s likely.

Recommendations

AES128 should be considered a default for symmetric encryption. AES256 should be used for higher security applications. *Full* Rijndael 256 can also be used (*full* meaning that the key as well as the block sizes are 256), however AES256 is looked at as the more accepted high security standard. Keys should be randomly generated. In situations where keys need to be derived from passwords, PBKDF2 should be used with strong salts and high stretching iterations.

CBC mode should be used unless there is a specific reason not to; this will rarely be the case. IVs should be randomly generated (they are by default). PKCS7 padding is recommended for its robustness.

To prevent padding oracle attacks, message authentication codes (MACs) should be used. These will be covered next chapter.

On a final note, we do not recommend using OTPs for data protection.

Chapter Summary

1. Symmetric encryption algorithms use the same secret key to encrypt and decrypt data. Keys should be randomly generated, and securely stored and transmitted.
2. Advanced Encryption Standard (**AES**) uses the **Rijndael** algorithm and is secure for use in modern production systems. **TripleDES** should not be used unless for legacy compliance. DES and RC2 should not be used under any circumstances.
3. CBC Mode is the most secure and robust cipher mode for use with the current .NET algorithms. For secure CBC Mode implementation, initialization vectors (IVs) must be randomly generated. While IVs do not need to be kept secret, they must be transmitted with the ciphertext to be used in the decryption process. .NET implementations of algorithms default to CBC mode but it’s always best to explicitly set it through the **Mode** property.
4. The most important characteristic of padding is being able to correctly remove it. PKCS7 is the recommended padding mode in .NET and is the default for .NET algorithms.
5. **CryptoStream** or **ICryptoTransform** objects can be used when writing encryption code. **CryptoStream** objects can process stream data and perform the underlying cryptographic operation. The **TransformFinalBlock** method of the **ICryptoTransform** interface will perform the cryptographic operation on the specified data and is simpler than a **CryptoStream** implementation for most solutions. However, larger resources such as big files or network streams should be processed by **CryptoStream** objects for easier handling and less memory usage.

6. One-time pads (OTPs) are theoretically unbreakable but introduce immense key management issues and are considerably less efficient than a block cipher. Generally, they are not recommended in production scenarios.
7. The most important aspect of symmetric key encryption is the strength and security of the private key. Cryptographic code must be carefully reviewed to avoid accidentally disclosing a key.

Chapter Questions and Exercises

1. What protections does symmetric encryption offer?
2. What is a block cipher mode? Which modes are available in .NET?
3. What symmetric algorithms does .NET implement? What is the strongest symmetric algorithm available in .NET?
4. What are the available key sizes for AES? Which is most secure?
5. What are the available key and block sizes for Rijndael? Which are the most secure?
6. Why is padding necessary? Which padding modes does .NET offer?
7. Explain what initialization vectors (IVs) do.
8. Write a program to encrypt and decrypt string data using AES256-CBC.
9. Write a program to encrypt and decrypt string data using AES256-CBC with a key derived from a Password-Based Key Derivation Function (covered last chapter).

Scenarios

1. You are expected to write a simple method for encrypting and decrypting string data. You don’t have to worry about generating keys or managing the data—your only task is to develop encrypt and decrypt methods that take the data and password as input. Passwords and data will be supplied as strings. You have to return data as a string type from the encrypt method and are expected to decrypt the same string data. What type of challenges might you expect? Does anything need to be clarified with the consumers of the methods or the data? Specifically, what type of formatting decisions will you have to make?

7 Message Authentication

Authentication: *A process that establishes the source of information, provides assurance of an entity's identity or provides assurance of the integrity of communications sessions, messages, documents or stored data.*

Chapter Objectives

1. Recognize the security risk posed by an attacker tampering with a message.
2. Understand how message authentication codes provide evidence of tampering.
3. Learn about different types of message authentication codes.
4. Learn how to create and verify message authentication codes using the **HMAC** base class.
5. Understand the Horton principle.
6. Learn how to implement an HMAC with a symmetric encryption algorithm such as AES.
7. Recognize the risks of truncating a MAC tag.

Encryption provides privacy but it lacks the means to guarantee integrity. An attacker who compromises a secret message by obtaining a key could view its private contents. An even bigger threat is represented by the attacker inserting his or her own data in place of the original. Another threat is presented where an attacker can simply tamper with the message to disrupt legitimate communications. Message Authentication is a way to verify that the message has not been changed (tampered with) or inadvertently corrupted since being secured, stored and transmitted.

The hash algorithms from last chapter can be used to check message integrity. A simple example would be hashing the plaintext message and attaching the hash to the ciphertext. Upon successful decryption, the message's hash could be compared with the attached hash to verify that it hasn't been changed in transit. The problem here is that an attacker who compromises the encryption key can simply rehash the message after making whatever changes he or she wants, and the recipient would be none the wiser. Another issue occurs where an attacker hashes a ciphertext and sends the unsecured hash with the packet. Here, an attacker could insert whatever message he or she wants, and its accompanying hash, and see if it works. These issues are what ultimately make the application of hash algorithms as discussed in chapter 4 unsuitable for the purposes of message authentication.

A different type of cryptographic primitive, a Message Authentication Code (MAC), is used to provide reasonable assurance that a message has not been tampered with (or indicate that it has been). A MAC uses a message and a symmetric key to produce a fixed-length output called a “tag”. Think of this functionality like a hash algorithm that also uses a key. The MAC tag is therefore a product of both the message and the secret key (in fact some MACs are known as *keyed hash algorithms*).

Once generated, a MAC tag is usually attached to the message it is being used to authenticate and they are persisted as a pair. This method allows the receiver, assumed to possess the symmetric authentication key, to verify that the received message generates a MAC tag identical to the one attached to it. If the result differs from the received MAC tag, this is indicative of the message being changed (tampered with) or using an incorrect key. Regardless, the verification process has failed and the message is not to be trusted.

In this chapter we will cover how MACs are used in secure communications and how they can be used with symmetric encryption algorithms. We will also cover some best practices and common security concerns.

Example: Best Practices Quick Start with HMAC-SHA256

Before we get into too many concepts with not enough code, here’s a quick example of MAC generation and authentication using a strong MAC algorithm called HMAC-SHA256. It’s built atop the SHA256 hashing algorithm and generates a 256-bit MAC tag (hash) from a variable-length input (the key as well as the message can be variable-length, but the key *should* be at least the length of the tag, in this case 256 bits).

Besides setting an authentication key, the generation uses the same **ComputeHash** method as the **HashAlgorithm** base class that we covered in chapter 4. The method below takes a key and a message as input and returns the MAC tag as output:

```
public byte[] GetHMACSHA256(byte[] key, byte[] message)
{
    using (HMACSHA256 hmacSha256 = new HMACSHA256(key))
    {
        return hmacSha256.ComputeHash(message);
    }
}
```

The next method is used to verify a message against a found/received MAC tag using an authentication key (the same key used to generate the alleged MAC). The message itself may look a little over-engineered compared to a simple byte array comparison method. The reason is that we have built it to take approximately the same amount of time to compare an invalid as it would take to compare a valid one. This helps defend against timing oracle attacks, which will be discussed later in the chapter.

```
public bool AuthenticateHMACSHA256(byte[] foundHMAC, byte[] message, byte[] key)
{
    byte[] computedHMAC = null;

    int mismatch = 0;

    using (HMACSHA256 hmac = new HMACSHA256(key))
    {
        computedHMAC = hmac.ComputeHash(message);
        hmac.Clear();

        if (foundHMAC.Length != hmac.HashSize >> 3) mismatch++;
    }

    for (int i = 0; i < computedHMAC.Length; i++)
    {
        if (foundHMAC.Length > i)
        {
            if (foundHMAC[i] != computedHMAC[i]) mismatch++;
        }
        else
        {
            mismatch++;
        }
    }

    return mismatch == 0;
}
```

Next, we'll try these methods out using an empty byte array as the message and a randomly generated key:

```
byte[] message = new byte[40];

byte[] key = new byte[32];

using (var rng = new RNGCryptoServiceProvider())
    rng.GetBytes(key);

byte[] macTag = GetHMACSHA256(key, message);

bool ok = AuthenticateHMACSHA256(macTag, message, key);
```

Block-Cipher MACs

Block-cipher MACs are usually used as modes of operation in a block cipher where the block cipher algorithm is used to generate the MAC tag. These also produce fixed-length output given a variable-length input but usually require a fixed-length key to be used with the symmetric algorithm (i.e. AES, Rijndael).

CBC-MAC

CBC-MAC uses the same block cipher that encrypted the data to produce a MAC. The most common implementation of this involves placing the cipher in CBC mode, setting the IV to a series of zeros, and encrypting the message with a secret authentication key. The last block of the encrypted result is used as the MAC. The primary problem with CBC-MAC is that it's easy to implement incorrectly. A secondary issue is that when used with a 128-bit block cipher (the standard), CBC-MAC only provides 64-bits of security under the generic attack models. Systems attempting to maintain a 128-bit security level would have to use a block cipher with key and block sizes of at least 256-bits. .NET only has one block cipher that could meet these criteria: Rijndael256.

The code below illustrates the functionality of CBC-MAC using full Rijndael 256. This method is a little on the home-grown side since we are actually using regular CBC mode and simply taking the last block of data rather than using a cipher mode that does this for us:

```
byte[] Rijndael256_CBC_MAC(byte[] data, byte[] key)
```

```
{
    using (RijndaelManaged rijn = new RijndaelManaged())
    {
        rijn.KeySize = 256;
        rijn.BlockSize = 256;
        rijn.IV = new byte[32];
        rijn.Key = key;
        rijn.Mode = CipherMode.CBC;

        using (ICryptoTransform encryptor = rijn.CreateEncryptor())
        {
            byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);

            rijn.Clear();

            return encryptedData.Skip(encryptedData.Length - 32).ToArray();
        }
    }
}
```

Hashed Message Authentication Codes (HMACs)

HMAC is a standardized method of generating Hashed Message Authentication Codes (HMACs). HMACs are a type of keyed hash algorithm and take two values as input: a variable-length message, and a secret key. The HMAC output, like the hash algorithms we covered previously, is fixed-length. Most HMAC implementations are built atop cryptographic hash functions such as SHA256. HMACs in most programming languages and frameworks provide a simpler interface than block-cipher MACs and as a result are more robust and less error-prone.

Authentication Keys

Authentication keys are just as important as symmetric encryption keys in terms of management. For instance, this means that you need to have a secure means for generation, distribution, and storage. Different types of MACs also have dissimilar requirements for authentication keys that must be considered.

The first question logically encountered by most developers is whether to use the encryption key as the MAC key. After all, block-cipher MACs will require the same key length for authentication as they do for encryption. It’s just as easy for HMACs since they have variable-length keys. However, the security risks this poses are actually quite a bit different between block-cipher MACs and HMACS. The most commonly implemented block-cipher MAC, CBC-MAC, should never be used with the same encryption key because it can leak information to an attacker. HMAC algorithms do not suffer from this particular issue. In either method, however, using the same key for authentication and encryption will give the attacker full control if either is compromised. Obviously if different keys are used for both encryption and authentication the attacker will have to compromise both to successfully read and tamper with the data.

Managing another key can pose a major change to the design of most secure solutions. Managing a single key is often intuitive when you look at the logistics of it being linked to a user password or account password. Key-exchange protocols give us a way to exchange or derive a symmetric key for encryption. But what about an authentication key? The aspects of not only introducing but also securely generating and protecting another key is usually an area where most developers drop the ball. Everything is in place for an encryption key, but seldom do designs set up a secondary secure generation and management system for an authentication key.

In many cases we see developers derive their authentication key from the encryption key or simply use the encryption key as the authentication key, which, of course, is a bad idea. The lesson: make sure you plan for how authentication keys will be generated and handled.

Another critical factor in MAC keys is length. For most block-cipher MACs this isn't an issue because they require a certain key size and this key size is secure for most modern block ciphers (128-bits or larger). HMACs are a little different because they can accept variable-length keys. HMAC keys should have a strength level equal to or greater than their output, or tag size. So, for a 256-bit HMAC, the key should be at least 256-bits.

Programming MACs in .NET

The **System.Security.Cryptography** namespace affords two options for message authentication: MACTripleDES, and HMACs. HMACs are recommended for both security and simplicity.

MACTripleDES

MACTripleDES derives from the **KeyedHashAlgorithm** base class and uses the **TripleDES** block cipher to encrypt the input data and produce a 64-bit MAC. **MACTripleDES** should not be used in modern secure applications because of its short MAC tag.

Hashed Message Authentication Codes (HMAC)

The **System.Security.Cryptography** namespace contains HMAC implementations of common hash algorithms like the SHA-2 series, all of which are concrete implementations of the **HMAC** class. Table 22 shows the different HMAC implementations in .NET and their respective key and hash sizes.

Table 22: HMAC implementations in .NET

HMAC Class Name	Base Class	Key Size	Hash Size (bits)
HMACMD5	HMAC	variable	128
HMACSHA1	HMAC	variable	160
HMACRIPEMD160	HMAC	variable	160
HMACSHA256	HMAC	variable	256
HMACSHA384	HMAC	variable	384
HMACSHA512	HMAC	variable	512

The HMAC Class

The **HMAC** base class has a straightforward functionality inherited from the **KeyedHashAlgorithm** (which in turn inherits from the **HashAlgorithm** class).

Creating an Instance

The factory-style static **Create** method can be used without arguments to create a default instance of **HMACSHA1**:

```
HMAC hmacSha1 = HMAC.Create();
```

Alternatively, a string argument can be supplied for the name of the algorithm to be created:

```
HMAC hmacSha256 = HMAC.Create("HMACSHA256");
```

A class instance can also be created directly using the **new** keyword:

```
HMACSHA256 hmacSha256 = new HMACSHA256();
```

Setting the Authentication Key

For an instance of HMAC to securely generate a hash, a key and a message are needed. The key must be set prior to generating a hash. This can happen two ways:

1.) Set the **Key** parameter of the **HMAC** class after creating an instance:

```
HMAC hmacSha1 = new HMACSHA1();

hmacSha1.Key=...
```

2.) Set the key through the constructor of an **HMAC** subclass, shown below with **HMACSHA256**. To be clear, the key cannot be supplied through the HMAC base class.

```
byte[] key = ...
HMACSHA256 hmacSha256 = new HMACSHA256(key);
```

Computing the HMAC Tag

The **ComputeHash** method is used to generate the MAC tag once the key has been set. Like the **HashAlgorithm** class, the **ComputeHash** method will work for almost all scenarios. This method takes a variable-length byte array of data to hash and returns a fixed-length byte array as the hash result. Overloads allow use of a byte array that specifies a particular index range to hash, or alternatively, a **Stream**.

Below, an example method specifies a key and a message as parameters and computes an HMAC tag using **HMACSHA256**:

```
public byte[] GetHMACSHA256(byte[] key, byte[] message)
{
    using (HMACSHA256 hmacSha256 = new HMACSHA256(key))
    {
        return hmacSha256.ComputeHash(message);
    }
}
```

A more generic HMAC method could be drawn up like this:

```
public byte[] GetHMAC<T>(byte[] data, byte[] key) where T: HMAC, new()
{
    byte[] hmac = null;

    using (T alg = new T())
    {
        alg.Key=key;
        hmac = alg.ComputeHash(data);
        alg.Clear();
    }

    return hmac;
}
```

Verifying a MAC Tag

Verifying a message's MAC tag is a straightforward process:

1. Retrieve the MAC tag from the received message.
2. Compute the MAC tag for the message.
3. Compare the retrieved MAC tag against the computed MAC tag. If the MAC tags match, the message is deemed authentic.

One consideration when authenticating a MAC is information leakage associated with the execution time of the authentication function. This type of vulnerability can lead to a Side-Channel Attack.

The following function was found on a popular programming forum. It performs a byte-by-byte comparison on two byte arrays (MACs) and returns false at the first byte mismatch:

```
private bool Compare(byte[] a, byte[] b)
{
    int mismatch = 0;

    for (int i = 0; i < a.Length; i++)
    {
        try
        {
            if (a[i] != b[i]) return false;
            else mismatch++;
        }
        catch
        {
            return false;
        }
    }

    return mismatch == 0;
}
```

This method was written sloppily by a lazy programmer trying to be efficient. CPU cycles are saved by not checking the rest of the bytes after the first mismatch, which could present a timing oracle attack. A closer look reveals that this is the least of the security issues in this method. Here, the programmer leaves a huge hole: The lack of any explicit checking of byte array length in this method means that if array *a* has a length of zero, this method will always return *true,* regardless of array *b's* value. The following use of the insecure *Compare* method shows how easy it is to exploit its poor construction and reliance on extraneous validation of the input:

```
byte[] a = new byte[0];
byte[] b = new byte[] { 1, 2, 3, 4 };

bool hasMatch = Compare(a, b); //returns true!
```

A Better HMAC Authentication Function

To prevent attackers from *gleaning* the type of information leakage associated with timing side-channel attacks, authentication functions should take the same time to compare a bad MAC as an authentic one. Methods can be constructed that provide a more consistent comparison time and disallow malformed input data from tricking the function into returning true.

Here is an example of a generic authentication function that can use any **HMAC** subclass as the generic type parameter and provides better and security than the *Compare* shown earlier. Notice also that the success of the function is dependent on the input HMAC value, *foundHMAC*, having the same length as the algorithm's hash size:

```
public bool AuthenticateHMAC<T>(byte[] foundHMAC, byte[] message, byte[] key) where T:
HMAC, new()
{
    byte[] computedHMAC = null;

    int mismatch = 0;

    using (T hmac = new T())
    {
        hmac.Key=key;

        computedHMAC= hmac.ComputeHash(message);

        hmac.Clear();

        if (foundHMAC.Length != hmac.HashSize>>3) mismatch++;
    }

    for (int i = 0; i < computedHMAC.Length; i++)
    {
        if (foundHMAC.Length > i)
        {
            if (foundHMAC[i] != computedHMAC[i]) mismatch++;
        }
        else
        {
            mismatch++;
        }
    }

    return mismatch == 0;
}
```

The function is not prematurely stopped for anything, and does not explicitly perform data validation. Whether this is good practice for cryptographic programming is debatable. Many functions would accept the null data and still try to deliver a consistent/flat time, because after all, the thrown error can lead to information leakage. The caveat in this case would be if the developer actually intends to allow null data or keys into their method. Here it is obviously allowed (.NET will still throw a **NullReferenceException**), but this really comes down to the level of abstraction that the developer is working with, measures implemented for error handling, and how much the data is being sanitized. Still, these are things to keep in mind.

Order of Authentication and Encryption

At some point when writing cryptographic application code the question will come about in which order encryption and message authentication should be performed. There are three main permutations for this and each has its pros and cons.

Option 1: Encrypt-then-Authenticate

Option 2: Encrypt-and-Authenticate

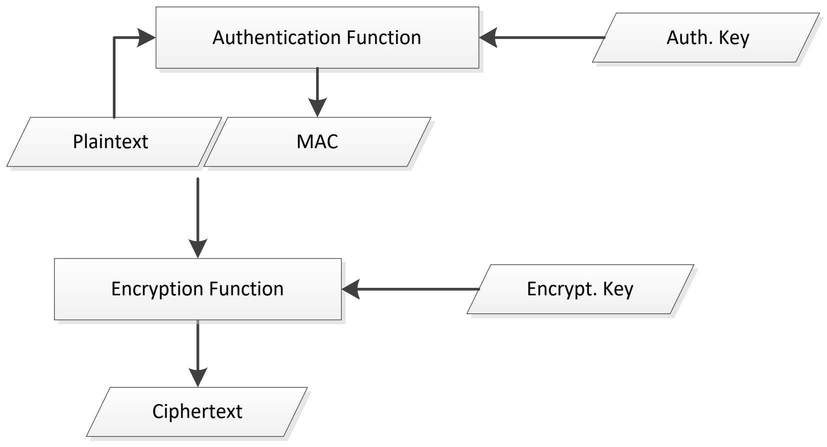
Option 3: Authenticate-then-Encrypt

The following examples will use the generic HMAC methods, *GetHMAC<T>* and *AuthenticateHMAC<T>,* that we created earlier in the chapter and the simple *Encrypt* and *Decrypt* methods that we used for AES256 on page 89.

Encrypt-then-Authenticate

Encrypt-then-authenticate is performed in the order it is spoken. *Encrypt* the secret message to generate the ciphertext, and *then authenticate* the ciphertext. In practice this would be performed by encrypting your message with a block cipher like AES and then passing the AES ciphertext through an **HMAC** like **HMACSHA256**. The HMAC hash would then be transmitted with the ciphertext so it can be authenticated by the recipient. Figure 25 illustrates encrypt-then-authenticate as the sender.

Figure 25: Encrypt-then-Authenticate (sender)

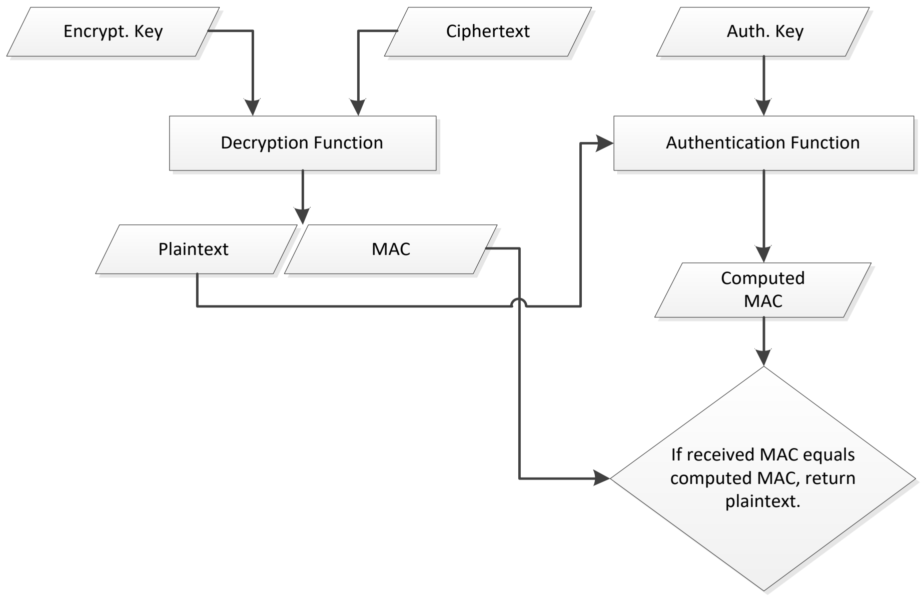


```
public byte[] EncryptThenMAC(byte[] data, byte[] encryptionKey, byte[] authenticationKey)
{
    byte[] cipherText = Encrypt(data, encryptionKey);

    byte[] hmac = GetHMAC<HMACSHA256>(cipherText, authenticationKey);

    return cipherText.Concat(hmac).ToArray();
}
```

The order is reversed on the recipient's end and results in a *Verify-and-Decrypt* process: The ciphertext and MAC tag are separated from each other, the ciphertext is removed and passed through the MAC function, and upon subsequent verification of the MAC the ciphertext can be decrypted. It is not required to perform this process in a particular order, but the order can have an effect on performance.

From a performance point of view, the authentication and decryption processes could be performed in parallel, but not on the encryption end where the ciphertext must exist before it can be authenticated. On the other hand, if authenticating then decrypting in a serial fashion, computing resources are expended to decrypt the ciphertext only if it can be successfully authenticated. Figure 26 illustrates encrypt-then-authenticate as the receiver.

Figure 26: Encrypt-then-Authenticate (receiver)



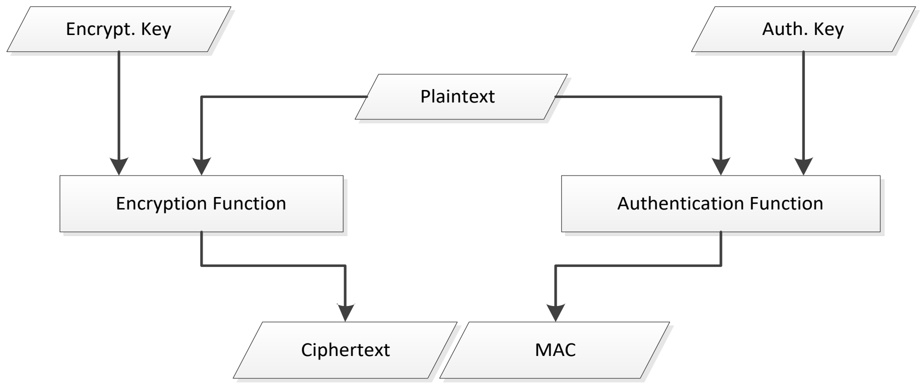
```
public byte[] DecryptThenAuthenticate(byte[] data, byte[] encryptionKey, byte[]
authenticationKey)
{
    byte[] foundHmac = data.Skip(data.Length - 32).ToArray();

    byte[] cipherText = data.Take(data.Length - 32).ToArray();

    byte[] plaintext = Decrypt(cipherText, encryptionKey);

    if (AuthenticateHMAC<HMACSHA256>(foundHmac, cipherText, authenticationKey))
    {
        return plaintext;
    }
    else
    {
        plaintext = null;
        return null;
    }
}
```

Encrypt-and-Authenticate

Encrypt-and-authenticate performs encryption *and* authentication on the plaintext. This is different from *encrypt-then-authenticate* where the MAC is a product of the ciphertext. The MAC is still attached to the ciphertext and transmitted to the recipient where the message is parsed and processed.

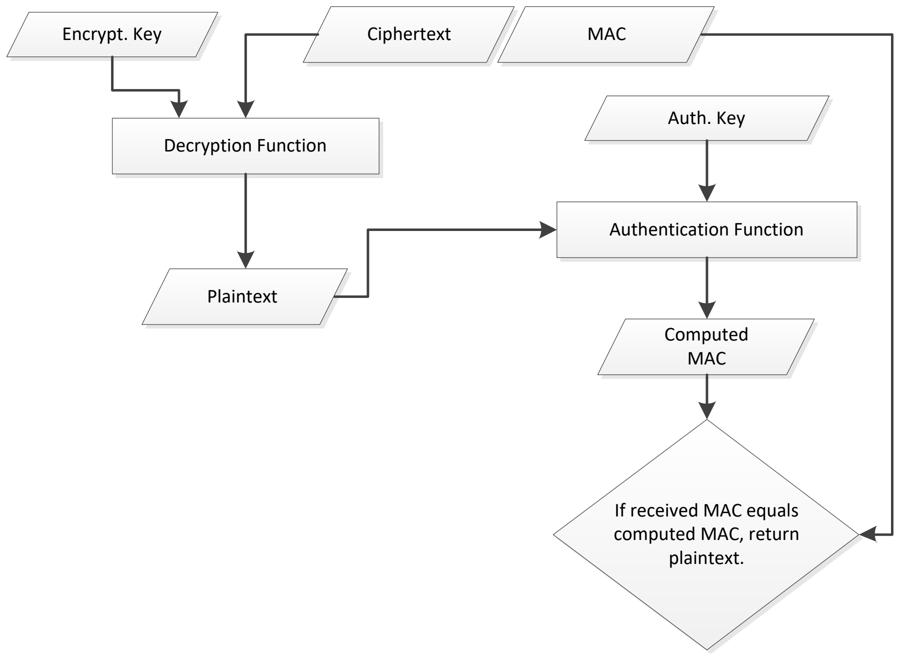
Encrypt-and-authenticate can be performed in parallel by the sender but not the receiver. The receiver also has to incur the possible performance issues from decrypting before authenticating. These amount to wasted computing resources spent on decrypting if the message does not authenticate. Figure 27 and Figure 28 illustrate the encrypt-and-authenticate process as the sender and receiver, respectively.

Figure 27: Encrypt-and-Authenticate (sender)



```
public byte[] EncryptAndMAC(byte[] data, byte[] encryptionKey, byte[] authenticationKey)
{
    byte[] cipherText = Encrypt(data, encryptionKey);

    byte[] hmac = GetHMAC<HMACSHA256>(data, authenticationKey);

    return cipherText.Concat(hmac).ToArray();
}
```

Figure 28: Encrypt-and-Authenticate (receiver)



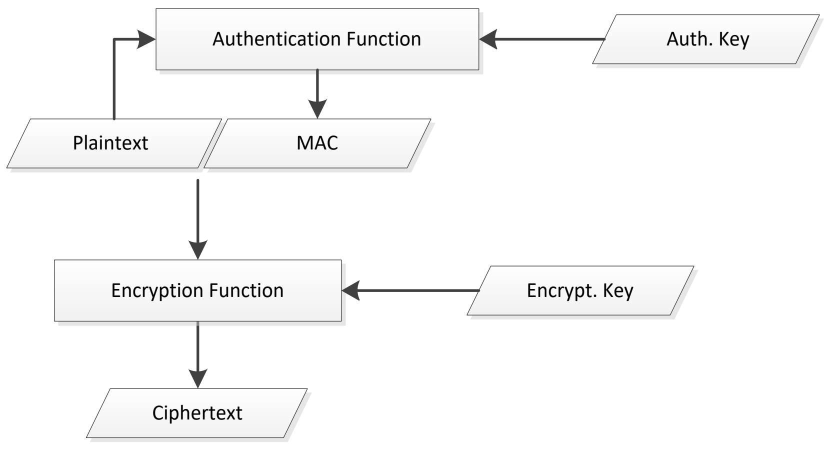
```
public byte[] DecryptThenAuthenticate(byte[] data, byte[] encryptionKey, byte[]
authenticationKey)
{
    byte[] foundHmac = data.Skip(data.Length - 32).ToArray();
    byte[] cipherText = data.Take(data.Length - 32).ToArray();

    byte[] plaintext = Decrypt(cipherText, encryptionKey);

    if(AuthenticateHMAC<HMACSHA256>(foundHmac,plaintext,authenticationKey))
    {
        return plaintext;
    }
    else
    {
        plaintext = null;
        return null;
    }
}
```

Authenticate-then-Encrypt

Authenticate-then-encrypt is different from the first two options because the MAC is not visible to the attacker. This is achieved by first generating a MAC tag from the plaintext, then encrypting the plaintext with the MAC tag attached (usually appended), and transmitting the concatenated result to the recipient. The recipient must decrypt the message before it can be parsed to remove and subsequently authenticate the MAC.

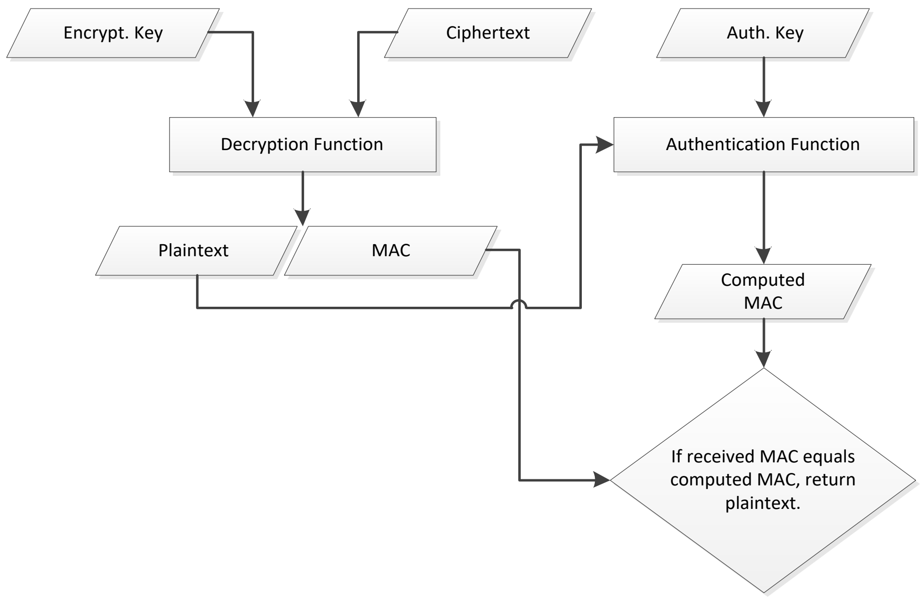
The nature of *authenticate-then-encrypt* removes the possibility of performing encryption/decryption and authentication in parallel. The sender must complete the MAC before encrypting, and the receiver must complete the decryption before authenticating. Like *encrypt-and-authenticate,* this can precipitate wasted CPU cycles (time) on decrypting messages that might not even authenticate. However, *encrypt-and-authenticate* can be performed in parallel on the sender's end. Figure 29 and Figure 30 illustrate the authenticate-and-encrypt process as the sender and receiver, respectively.

Figure 29: Authenticate-then-Encrypt (sender)



```
public byte[] MACThenEncrypt(byte[] data, byte[] encryptionKey, byte[] authenticationKey)
{
    byte[] hmac = GetHMAC<HMACSHA256>(data, authenticationKey);

    byte[] plaintextAndHmac = data.Concat(hmac).ToArray();

    return Encrypt(plaintextAndHmac, encryptionKey);
}
```

Figure 30: Authenticate-then-Encrypt (receiver)



```
public byte[] DecryptThenAuthenticate(byte[] data, byte[] encryptionKey, byte[]
authenticationKey)
{
    byte[] plaintextAndHmac = Decrypt(data, encryptionKey);

    byte[] plaintext = plaintextAndHmac.Take(plaintextAndHmac.Length - 32).ToArray();

    byte[] foundHmac = plaintextAndHmac.Skip(plaintext.Length).ToArray();

    if (AuthenticateHMAC<HMACSHA256>(foundHmac, plaintext, authenticationKey))
    {
        return plaintext;
    }
    else
    {
        plaintext = null;
        return null;
    }
}
```

Return null or Throw an Exception?

The example code last section illustrates the different orders or authentication and encryption using symmetric encryption algorithms and HMACs. In our examples we return null where the HMAC verification fails. We also could have thrown an exception. The choice is really up to the programmer and what behavior they want to be exhibited. In both cases it's a good idea to log the failure and the source of the possible attack.

Which Order is Most Secure?

Valid arguments can be made for each method and use case. *Authenticate-then-encrypt* doesn't expose the MAC to the attacker, thereby forcing them to break the ciphertext first. In most cases, this is considered a more secure option. The MAC tag is also a product of the plaintext, not the ciphertext, which better adheres to the Horton Principle. *Encrypt-then-authenticate* is the most vulnerable to tampering; it exposes the most information to the attacker since the MAC is derived from the ciphertext, and the MAC is also visible.

The examples in this book will use *authenticate-then-encrypt* and *encrypt-and-authenticate*, with the recommendation that an *authenticate-then-encrypt* scheme is best for developers trying to expose little to the attacker and achieve the best tamper resistance.

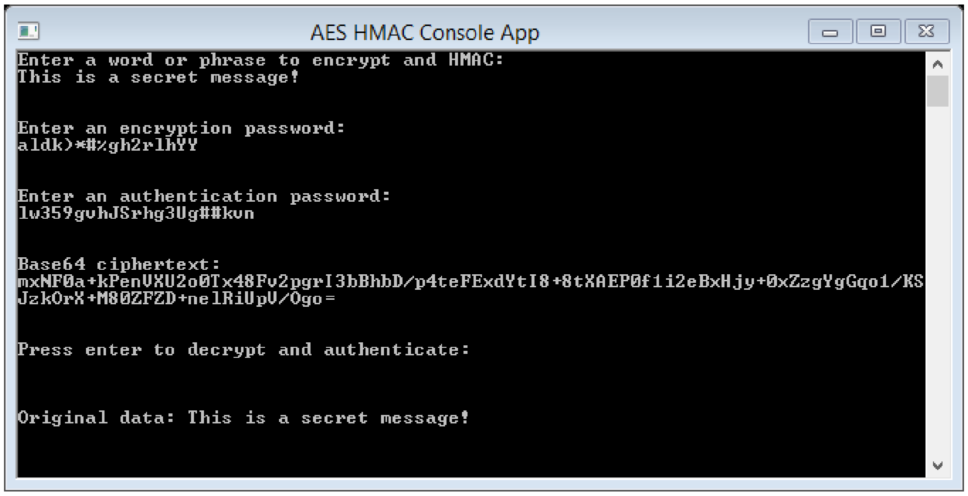
Example: AES256 with HMACSHA256

In this example we use AES256 and HMACSHA256 in a simple console app that encrypts and decrypts data. This example will just be a little more expanded version of the authenticate-then-encrypt example from last section. We use the *Encrypt* and *Decrypt* methods from page 89, the generic *GetHMAC<T>* and *AuthenticateHMAC<T>* methods from earlier in this chapter, and the *MACThenEncrypt* and *DecryptThenAuthenticate* methods from last section (flip back to these sections for the code). **SHA256Managed** is used to derive 256-bit keys out of the user passwords to use with the **AesManaged** and **HMACSHA256** classes. You'll also notice that we have a full example of text encoding and Base64 conversion for handling string data.

```
static void Main(string[] args)
{
    Console.Title = "AES HMAC Console App";

    Console.WriteLine("Enter a word or phrase to encrypt and HMAC:");

    string message = Console.ReadLine();

    Console.WriteLine("\n\nEnter an encryption password:");

    string encryptionPassword = Console.ReadLine();

    Console.WriteLine("\n\nEnter an authentication password:");

    string MACPassword = Console.ReadLine();

    byte[] encryptionKey;
    byte[] MACKey;

    using (SHA256Managed sha256 = new SHA256Managed())
    {
        encryptionKey = sha256.ComputeHash(Encoding.UTF8.GetBytes(encryptionPassword));
        MACKey =sha256.ComputeHash(Encoding.UTF8.GetBytes(MACPassword));
        sha256.Clear();
    }

    byte[] ciphertext = MACThenEncrypt(Encoding.UTF8.GetBytes(message), encryptionKey,
MACKey);

    string b64ciphertext = Convert.ToBase64String(ciphertext);

    Console.WriteLine("\n\nBase64 ciphertext:\n" + b64ciphertext);

    Console.WriteLine("\n\nPress enter to decrypt and authenticate:");

    Console.ReadLine();
    byte[]
plaintext=DecryptThenAuthenticate(Convert.FromBase64String(b64ciphertext),encryptionKey,MAC
Key);

    Console.WriteLine("\n\nOriginal data: " + Encoding.UTF8.GetString(plaintext));

    Console.ReadLine();
}
```

Figure 31 shows a demo of the AES HMAC console app.

Figure 31: The AES HMAC Console App



```
Enter a word or phrase to encrypt and HMAC:
This is a secret message!


Enter an encryption password:
aldk)*#%gh2rlhYY


Enter an authentication password:
lw359gvhJSrhg3Ug##kvn


Base64 ciphertext:
mxNF0a+kPenVXU2o0Tx48Fv2pgrI3bBhbD/p4teFExdYtI8+8tXAEP0f1i2eBxHjy+0xZzgYgGqo1/KS
JzkOrX+M80ZFZD+nelRiUpV/Ogo=


Press enter to decrypt and authenticate:


Original data: This is a secret message!
```

The Horton Principle

The Horton Principle is highly applicable to secure application development and says that authentication should include more than the data itself, it should include the context. An example of this is authenticating everything of importance that is associated with the message; this can include protocol version numbers, times, counters/message numbers, and anything that could be tampered or spoofed. Increasing the scope and amount of information that is authenticated makes the attacker's job harder and cannot help but make for a more secure system.

The Horton Principle also has a place in networking and the developer's level of abstraction in relation to the network stack. For example, just because TCP ensures a correct delivery of data at layer 4, or because SSL has given us a secure connection, does not mean that this should be considered adequate message authentication for a ciphertext that we are using in an application. Context is what matters here. Is the data being authenticated in the context in which it is meant to be used? Does the process of authentication actually verify that the message and all other relevant data associated with it are correct *in the context they are decrypted in*?

This begs the question: What additional things should a developer decide to authenticate and how are these things verified? This is a two-part problem. First, the developer has to look at what things actually authenticate the message and *what it means*. Second, there must be a system in place to verify the additional data, in which each endpoint has to be aware of the data being transmitted and how to verify it. MACs are simple, the message itself is transmitted, so naturally the means are available to authenticate as long as there is a shared key and the process (the algorithm) is known by both parties. In reality, authenticating much more data than the message will still come down to a process created by the developer and distributed to all parties involved.

Like many other aspects of security, choosing what to authenticate can come down to usability vs. security. The more data that must adhere to strict authentication, the smaller the window the attacker has to work within, but the same usually goes for legitimate users. The list below contains some aspects of communications and communication channels that can be authenticated to enhance security.

- Message Number or Sequence Number
- Nonce
- Message Length
- Padding
- Protocol Version Numbers
- GPS and Geolocation
- Dynamically Generated Tokens
- Date and Time
- Timeframe for Receipt of Message
- Digital Signature
- Machine or system identifiers such as IP, subnet, ports numbers, MAC Address, or device ID should be included for source and destination.

The Storage Cost of Message Security

The amount of data being stored and protected is growing. As a corollary, companies' costs are increasing as they must house this data and maintain high availability. Encrypted and hashed data can require more storage space depending on the format in which it is stored and how it has been encrypted. Base64 encoding is used for storing or transmitting encrypted data in a text or string format and will add 1/3 more size to the payload than raw bytes.

Most encryption functions will not provide a perfect one-to-one mapping of plaintext bytes to ciphertext bytes, either. Padding can increase the length of a ciphertext between a byte and an entire block's length. Initialization vectors will usually add a block's length. MACs will add even more length. HMACSHA256 for example will add an additional 256 bits (32 bytes).

A 17-byte message encrypted with AES in CBC mode and authenticated with HMACSHA256 could expect quite a bit of overhead:

Plaintext=17 bytes

Padding=15 bytes

IV=16 bytes

HMAC=32 bytes

Ciphertext=80 bytes

This particular scenario required almost 5x more storage space for the ciphertext. Bigger data, however, will usually incur less of a storage hit in terms of percentage or ratio (if using Base64 encoding, there is no way to get around the extra 33%). Here's another example:

Plaintext=1025 bytes

Padding=15 bytes

IV=16 bytes

HMAC=32 bytes

Ciphertext=1088 bytes

Encrypting the 1025-byte plaintext added another 63 bytes for a total size of 1088 bytes. Like the last example, 63 bytes were added, but here it only required an increase of about 6%. It's not hard to see that encrypting lots of small pieces of data on a large scale could be huge in terms of storage costs, while larger pieces of data see less of an impact.

Truncating MACs

It's easy to find truncated MACs in the wild where developers are trying to save space. You'll commonly see 64-bit and 128-bit MACs truncated to 32-bits. In other cases you might see MACs that are truncated to 128 bits down from a 256 or 512-bit HMAC. These scenarios really aren't good or bad because we lack context. Truncating a MAC is a valid solution to some problems where it doesn't diminish the *expected* security. Keep in mind, too, that collision attacks apply to MACs, rendering their bit strength to half of their output, meaning a 256-bit MAC only provides 128-bit protection. Truncating a MAC down to 32 bits is irrelevant where we expect only 16 bits of protection. If you are trying to maintain 128 bits of tamper resistance you should be using a full 256-bit MAC, which is the recommendation for systems aiming for a true 128-bit security level. If you want 32 bits of tamper resistance you'll need a 64-bit MAC.

As a blanket policy we recommend 128 bits of tamper resistance using a 256-bit MAC (even if it's truncated from 512 bits of output). But again you have to look at the context. In scenarios where we accept the weakness of 16 bits of tamper resistance, a 32-bit MAC is not at all unreasonable.

Tamper-Evident is Not Tamper-Proof

In Secrets and Lies, Bruce Schneier emphasized a key point: we don't have anything that is tamper-proof; and most things that claim to be *tamper-proof* or *tamper-resistant* are actually *tamper-evident*.

Tamper-proof means that something cannot be tampered with. *Tamper-resistant* means something is resistant to being tampered with. In all practicalities these equate to the same thing: an attacker cannot effectively tamper with a target. The concept of tamper-proof or tamper resistance per se, within the contexts of most cryptographic applications, or really anything, is a misnomer. If our goal is to disrupt or interfere with the confidentiality, integrity, or availability of a particular object, then the concept of tamper-proof is dependent on having zero access to the object—no contact. Once we can touch it we can tamper with it. Any level of access, besides no access, opens the door to tampering.

We can use hardware as an example of this tampering issue. ABC Chip Co. specializes in manufacturing highly secure hardware, specifically chips. ABC claims their new X9000 chip is tamper-proof. Its pamphlet lists an abundance of rich new security features. ABC hires an outside penetration tester, Mr. Jack Burton (who also works part time as a long haul truck driver), to test the chip and assess its security, particularly in an attempt to tamper with the chip. So, Jack Burton goes to his lab, puts the chip in a static free environment, and then hits it with a 26 lb. sledge hammer. Voila! He successfully tampered with the chip. He then writes out his invoice to ABC Chip Co. for a $20,000 penetration test.

Maybe that wasn't the most practical example, but you get the point. Tamper-proof isn't a practical property for something to have if it's going to be implemented in a production environment. It's for this reason that the industry uses cryptographic primitives that are *tamper-evident* rather than tamper-proof. *Tamper-evident* means that we are able to verify with reasonable certainty that something has or hasn't been tampered with. Cryptographic primitives like Message Authentication Codes (MACs) and digital signatures are used to provide tamper evidence.

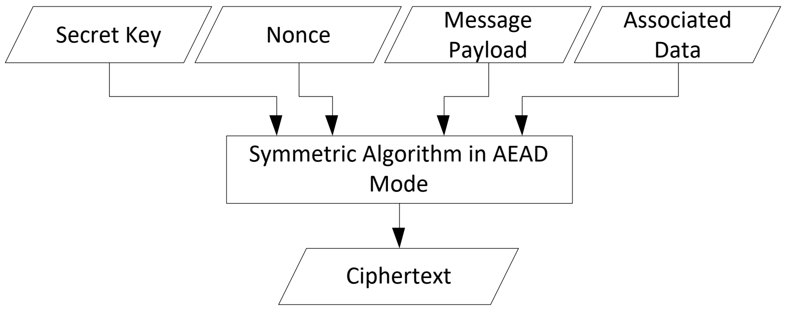
Tamper-Evidence: *MACs and digital signatures do not prevent attackers from tampering with data. They simply give the programmer or end user the means to guarantee the integrity of such data. The verification process for MACs and digital signatures is designed to provide a deterministic mechanism to indicate that the data either has or hasn't been tampered with.*

At the end of the day we can't stop an attacker from tampering with something if they have access to it. We can, however, implement solutions that notify us of such attacks and are capable of taking preventative measures.

Authenticated Encryption with Associated Data (AEAD)

Authenticated Encryption with Associated Data (AEAD) is a style of *authenticated encryption* using a symmetric algorithm in which the cipher mode performs the data protection and authentication (MAC). AEAD modes are fundamentally different from the cipher modes and MACs we've previously covered. One of the main differences between AEAD modes and the modes we've used in .NET is the number of input fields to the mode itself. Besides the key and message, these modes usually accept a nonce (number used once) and associated data. The associated data is authenticated but not protected. Figure 32 illustrates the input and output of a typical AEAD mode.

Figure 32: Typical AEAD Mode of Operation



Counter Mode with CBC-MAC (CCM) and Galois/Counter Mode (GCM) are popular AEAD modes that have been standardized by NIST, but are not included in the .NET framework. However, many third-party libraries implement these modes and there's a good chance you may end up using one.

Problems with Nonce-Dependent MACs

Some MACs need a nonce (a number only used once). While these are not part of the .NET **System.Security.Cryptography** library, some are popular in third-party libraries and popular in general. Proponents will usually advocate these MACs' security over that of the traditional variety; however, developers have a bad habit of reusing nonce values. GMAC is a good example.

GMAC is a NIST-standardized message authentication function that is designed for 128-bit block ciphers. GMAC is thrown around in forums and listed under product features as a secure way to provide message authentication. Besides some hype, GMAC is popular because it's efficient in hardware and software, and when used properly is a reasonably secure message authentication mechanism.

A GMAC takes three values as input, rendering it fundamentally different from other authentication functions like CBC-MAC, CMAC, and HMAC. GMAC needs a key, a message, and a nonce (used to generate an IV for the GMAC).

The critical breakdown in GMAC security occurs upon nonce reuse. Because nonce reuse in GMAC relates directly to IV reuse, it exposes a vulnerability that can cripple the entire function. In fact, the issue of nonce control in GMAC can't even be looked at as a best practice—*it's a must practice* (security in wireless WEP encryption is crippled by an IV issue).

This problem isn't even with GMAC, per se, or its process, but rather with how developers implement it. Most developers won't do what is necessary to prevent GMAC from breaking down. The extra measures that must be correctly introduced to handle nonce generation and control make GMAC a more expensive and error-prone implementation than simpler solutions like HMAC or CMAC. All too often developers have little incentive to deal with the kind of hassle and additional overhead that secure nonce management can present. Therefore, solutions that run GMAC without a secure nonce system should be presumed insecure and avoided.

Recommendations

Traditionally developers place far less emphasis on the strength of a MAC than that of the encryption algorithm, that is, if they even use a MAC. This is bad security as the MAC adds a critical layer of tamper evidence that defends against attacks that change the encrypted data. So first, the choice must be to actually use a MAC. Then, the MAC must be selected.

CBC-MAC requires getting a number of implementation issues correct to maintain security and if implemented incorrectly can also degrade message security. It also requires a 256-bit block cipher to maintain a 128-bit security window under generic attack models. **MACTripleDES** falls too short in the size category to be a reasonable candidate. In .NET, this leaves the HMAC implementations. HMACSHA256 is recommended for 128-bit security.

Chapter Summary

- Encryption provides confidentiality but cannot indicate if an attacker tampers with the data. Message authentication is the process of confirming the integrity of encrypted data.
- Message Authentication Codes (MACs) are used to enforce integrity in the message authentication process. There are different kinds of MACs, the most common being block-cipher MACs and hashed message authentication codes (HMACs).

- HMACs are recommended for their simplicity, security, and robustness. HMACSHA256 is recommended for modern production systems.
- The Horton Principle illustrates the many aspects of communication channels that can and should be considered for authentication.
- The order of encryption and authentication can place an emphasis on different aspects of security.

Chapter Questions and Exercises

1. Explain the purpose of MACs. How are they different from the hash functions we've covered in previous chapters?
2. What is the difference between block-cipher MACs and HMACs?
3. Explain the Horton principle. Which data should be authenticated?
4. What are the different orders of authentication and encryption? What are their advantages and drawbacks?
5. How do you determine a secure key length for HMAC keys?
6. Which HMAC algorithms are offered in .NET? Which are recommended?
7. Explain the risks of truncating a MAC.
8. What are AEAD modes? How are they different from regular MACs?
9. Write a program that encrypts and authenticates a message using an *authenticate-then-encrypt* order (you should also perform a decryption and verification process in the program).

Scenarios

1. You are building a secure application for a client who wants 128-bit security strength. The client asked a friend of theirs, a security expert, which algorithms should be used for keeping their data safe. Their friend recommended AES (128-bit) and a HMACMD5 (128-bit tag). Is this combination able to provide a 128-bit security strength? Why or why not? What would you tell the client?

8 Asymmetric Encryption and Key Exchange in .NET

Asymmetric Algorithm: *A cryptographic algorithm that uses two related keys, a public key and a private key. The two keys have the property that determining the private key from the public key is computationally infeasible. Also known as a public key algorithm.*

Chapter Objectives

1. Understand the asymmetric model and how it is used.
2. Identify the limitations of asymmetric algorithms.
3. Learn how to use the RSA and Diffie-Hellman implementations in .NET and identify the differences in their usage.
4. Learn how to work with asymmetric keys using CspParameters and CngKey objects.
5. Know how to encrypt and decrypt data using RSA subclasses and key exchange subclasses.
6. Recognize the benefit of the hybrid encryption model (combining asymmetric and symmetric encryption).

Asymmetric encryption is one of the most notable developments in cryptography because it solves the issues associated with key distribution and key negotiation. It also makes available some other benefits like nonrepudiation which will be discussed with digital signatures.

The biggest problem with trying to communicate with parties via symmetric key encryption (like AES) is generating and negotiating keys in a secure manner between all the parties. If the communication protocol is insecure and cannot be trusted, then how can you be expected to send someone a key? This is where asymmetric cryptography comes in.

The Asymmetric model is most often used to negotiate symmetric keys over an insecure channel. This is due to asymmetric encryption being more computationally intensive than symmetric encryption, which renders it unsuitable for use in most applications as a primary means of encryption.

RSA

In 1978, Ronald Rivest, Adi Shamir, and Len Adleman invented what is now the most widely used public-key algorithm: RSA. .NET implements RSA through the **RSACryptoServiceProvider** and **RSACng** classes. Both implement the functionality of the **RSA** abstract base class.

RSACryptoServiceProvider uses the **CspParameters** object to store and retrieve keys from the CryptoAPI key store. **RSACng** uses a **CngKey** object to access the CNG key store. These key objects have much different interfaces and functionality despite the similarity between the **RSA** subclasses.

The RSA Base Class

The examples in this section will use the RSA base class to perform simple tasks, such as encryption and signing. The **RSA** base class has undergone some changes in the latest versions of .NET (4.6.1 is current as of this writing). Originally, the **RSA** base class had limited functionality to that of **RSACryptoServiceProvider**. The base class functionality has been substantially expanded and incorporates the new **RSAEncryptionPadding** and **RSASignaturePadding** objects into the encryption and signing methods, as well as other additions.

Creating an Instance

The factory-style static create method in the RSA class will create an default-configured instance of the RSA class. Currently, this is **RSACryptoServiceProvider** (check key sizes on defaults; they are frequently well below recommended sizes):

```
RSA rsa = RSA.Create();
```

Alternatively, for control over the derived type being created (**RSACryptoServiceProvider**, or **RSACng**) the **new** keyword can be used to instantiate a derived class directly:

```
RSA rsa = new RSACng();
```

Or:

```
RSA rsa = new RSACryptoServiceProvider();
```

Simple Encryption and Decryption

The **Encrypt** and **Decrypt** methods can be used for simple encryption and decryption. Byte-array data in addition to padding will have to be supplied for each of the methods. Padding will have to match for decryption to complete successfully.

Here’s a quick example:

```
RSA rsa = new RSACng();
```

```
byte[] data = new byte[32];

byte[] ciphertext = rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA1);

byte[] plaintext = rsa.Decrypt(ciphertext, RSAEncryptionPadding.OaepSHA1);
```

The above example uses a randomly generated Cryptography Next Generation (CNG) key pair, and encrypts an empty byte array using Optimal Asymmetric Encryption Padding (OAEP). **OaepSHA1** tells us that the padding is configured for use with the **SHA1** hash algorithm.

If you are familiar with the older style of specifying padding in **RSACryptoServiceProvider—**supplying a bool value to indicate OAEP padding or PKCS1—this will still work in the **RSACryptoServiceProvider** class even though it's not supported in the **RSA** base class. Table 23 describes each of the **RSAEncryptionPadding** members.

Table 23: Description of RSAEncryptionPadding Members

RSAEncryptionPadding Member	Description
CreateOaep(string)	Creates an object that represents OAEP padding mode for use with the supplied string algorithm name.
OaepSHA1	Creates an object that represents OAEP padding mode for use with SHA1.
OaepSHA256	Creates an object that represents OAEP padding mode for use with SHA256.
OaepSHA384	Creates an object that represents OAEP padding mode for use with SHA384.
OaepSHA512	Creates an object that represents OAEP padding mode for use with SHA512.
Pkcs1	Creates an object that represents PKCS #1.

RSACryptoServiceProvider

RSACryptoServiceProvider was the first implementation of RSA in the **System.Security.Cryptography** namespace. .NET 4.6 expanded its functionality under the **RSA** base class where the new **RSAEncryptionPadding** objects can be used to specify padding.

Creating an Instance

Like most of the other algorithms in this book, **RSACryptoServiceProvider** can be instantiated directly using the **new** keyword (the preferred method):

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
```

RSA.Create() will also create an instance of the default RSA implementation. Currently **RSACryptoServiceProvider** is the default for RSA in .NET. So, this could also create an instance (however, more explicit instantiation should be used):

```
RSA r = RSA.Create();
```

Encryption and Decryption

You can encrypt data using **RSACryptoServiceProvider** following practically the same steps that we showed in the **RSA** base class example:

```
RSA rsa = new RSACryptoServiceProvider();

byte[] data = new byte[32];

byte[] ciphertext = rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA1);

byte[] plaintext = rsa.Decrypt(ciphertext, RSAEncryptionPadding.OaepSHA1);
```

As we mentioned before, the **RSAEncryptionPadding** class is new to .NET 4.6 and older solutions that use **RSACryptoServiceProvider** may have to supply padding using the old bool style. This functionality is still available in .NET 4.6 if you declare and instantiate an **RSACryptoServiceProvider** instance (rather than declaring **RSA** and instantiating **RSACryptoServiceProvider** as we did in the above example):

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
```

The older style padding is set using a bool value as the second argument of the **Encrypt** or **Decrypt** methods. True specifies OAEP padding; false specifies PKCS#1 v1.5. Below we encrypt the same data as before specifying OAEP padding:

```
byte[] data = new byte[32];

byte[] ciphertext = rsa.Encrypt(data, true);

byte[] plaintext = rsa.Decrypt(ciphertext, true);
```

Before OAEP, PKCS#1 was the primary method of padding for asymmetric encryption. PKCS#1 is considered less secure than OAEP and its use should be limited to legacy compatibility. It’s recommended to use the newer style of padding if you have the option, unless you’re facing compatibility issues.

Specifying a Key Size

The default constructor for **RSACryptoServiceProvider** will create a new key pair using 1024-bit keys. There are a couple issues with this. First, 1024-bit keys are not especially strong and are inadequate for securing strong symmetric keys like those for AES (2048-bit keys should be used at a minimum). Second, .NET will create a new key pair every time an instance of **RSACryptoServiceProvider** is created unless a **CspParameters** object is used with an non-empty container, which can add additional overhead, especially if larger keys are being used (using **CspParameters** will be covered later this chapter).

Developers wanting to beef up their key sizes can specify the key size for the keys in the overloaded constructor. Legal key sizes range from 384 to 16384 bits. An instance is created below with 2048-bit keys:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(2048);
```

Importing and Exporting Keys

As stated last section, RSA will generate a new key pair upon construction unless a non-empty **CspParameters** object is being used in the constructor. This behavior will work for many applications where RSA keys are only being used temporarily to negotiate a symmetric key, or are being stored upon creation. Other circumstances will call for a specific RSA key or key pair. For these situations RSA can export keys that need to be stored or transmitted, as well as import keys under the same conditions. This functionality allows Bob to export his public key and send it to Alice, who would then import his public key. The simplicity of RSA in .NET and its effectiveness are largely the product of a well-designed import and export system for keys. Developers can control whether a key pair, or just the public key, are persisted, and in which format.

Methods used to export key information include: **ToXMLString**, **ExportRSAParameters**, and **ExportCspBlob**. Each of these methods allow the developer to control whether the private key information is exported. It's imperative that this is correct in a production environment as you will usually only want to transmit the public key to another party.

CspBlob

The most basic container for key material is a byte array. This option will remove many of the properties and methods that key container objects like **RSAParameters** and **CspParameters** allow the developer to access. On the flip side, byte arrays are easy to transmit and store, and are less of a hassle than trying to correctly serialize and deserialize complex objects.

ExportCspBlob will export key material in a byte array format from the RSA instance. This method takes a bool argument indicating whether the private key information will be exported. The example below generates a new key pair when the **RSACryptoServiceProvider** is created and exports the public key information as a byte array:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

byte[] publicKeyBlob = rsa.ExportCspBlob(false);
```

If the public and private key information needs to be exported, the bool argument in **ExportCspBlob** should be set to true:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

byte[] publicAndPrivateKeyBlob = rsa.ExportCspBlob(true);
```

Importing keys is just as simple as exporting. Importing byte array key information is performed through the **ImportCspBlob** method. This takes a byte array key blob as its only argument and does not need additional information to indicate if it contains a private key. Keys must be imported prior to using the instance to perform cryptographic operations such as encryption or decryption.

```
byte[] keyBlob = ...

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

rsa.ImportCspBlob(keyBlob);
```

RSAParameters

RSAParameters contains RSA key information for use in .NET. Like the previous examples it holds the public key with the option to include private key material. **RSAParameters** is serializable and contains eight properties relating to the composition of the key material. Luckily, developers rarely have to interact with these properties when using an **RSAParameters** object. After all, the point of the object is to encapsulate this information so the developer doesn't have to manage it at a micro level. However, while the developer doesn't normally see this data when dealing with **CspBlob** or XML strings, these objects do in fact contain all of the information stored in the **RSAParameters** properties.

Importing and exporting keys using **RSAParameters** is straightforward and follows the same logic as using the **CspBlob** objects from last example.

ExportParameters performs the export. The requisite bool argument from the other examples is used in the same manner to control the export of the private key info.

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
```

This exports only the public key:

```
RSAParameters publicParameters = rsa.ExportParameters(false);
```

This exports public and private keys:

```
RSAParameters publicAndPrivateParameters = rsa.ExportParameters(true);
```

ImportParameters imports an **RSAParameters** object:

```
RSAParameters parameters = ...

rsa.ImportParamters(parameters);
```

XMLString

XML can be handy. It can also be a burden. With the push toward JSON-formatted data in recent years by key players in the industry, XML in general is used less often but still has a firm grip in Microsoft technologies. Naturally, **RSACryptoServiceProvider** will import and export keys as XML strings. Exporting works like the previous examples through a method called **ToXMLString**.

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

string XMLPublicKey = rsa.ToXmlString(false);
```

Supplying the false argument in **ToXmlString** exports only the public key information. The XML string from the code above is formatted as follows (expect your modulus and export data to be different than the example):

```
<RSAKeyValue>
<Modulus>2OJmTc4EbNFid8cj6J5ZbqrWN54eGfzgCVInnTacGQUIChPm7InG8ULBp5m0EuErByORGmGajaNzGDH4Ns
ETODx+u/0GxiaYB+l6HyIoMqpUYxT4BYW3njh8jRUtIRluCLBXFc0PrmIt/QFFyOEL9+gvtJX0iKoy3i3f5J1M4gM=
</Modulus>
<Exponent>AQAB</Exponent>
</RSAKeyValue>
```

Exporting private key information adds much more to the XML payload:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

string XMLPublicAndPrivateKeys = rsa.ToXmlString(true);
```

```
<RSAKeyValue>
<Modulus>2OJmTc4EbNFid8cj6J5ZbqrWN54eGfzgCVInnTacGQUIChPm7InG8ULBp5m0EuErByORGmGajaNzGDH4Ns
ETODx+u/0GxiaYB+l6HyIoMqpUYxT4BYW3njh8jRUtIRluCLBXFc0PrmIt/QFFyOEL9+gvtJX0iKoy3i3f5J1M4gM=<
/Modulus>
<Exponent>AQAB</Exponent>
<P>9s21i9o7SFJBysZzhY0DcWP5RTzqC7VrLNkgahcvnpuvB1N54Y1eK62TRFlMR1+KODGdGl6l1oW8G28kASXGKQ==
</P>
<Q>4PdJnkwiOaWNBOt/8FaR8C4m/fLk8dSPb8ooPqDKfshmYbvC6S4SHwzLx53D0+wMSSrNaTsXGPmToRxqr4+0Sw==
</Q>
<DP>iqTk57ugsfADpbX2D4A3/ur6jTq6//jaTEdtPivoRGGQ4byzK1IPJNpNcIf5od659vdoGfgxkWFvWroEr+BYgQ=
=</DP>
<DQ>zrNn+QKLD9yEzdh0HSftv3koan0azvg3MsfUYnbql8MaDwKt/AJQtCbVtfvHSpjAURn60o1wk4n9kzLA875eMQ=
=</DQ>
<InverseQ>TXtt8vC3yYnVu5IIFFYae9cBKqL1yJL4AH6qWnea5N1K6mUITH5XfYra2K7gjHGOEwy1OO2hKUaWl6d8a
bADVQ==</InverseQ>
<D>GfSGL7t+9hLcyN7RIk6I/2B8gG1wxsVnflYRnZPifHAItUQKd7ZJU6gcitUvIq9FufX8sH6Lw3WTGYspXEwg43qM
a9eEuaEx94NpXCiikXwam2qFnlkPtSu6haDBKGgzPKUEKSlJMaEW2gl4YU5xJnbqxEuMCCOQx3QylpHY0LE=</D>
</RSAKeyValue>
```

FromXMLString imports RSA keys in typical fashion:

```
string XMLPublicKey =  ...

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

rsa.FromXMLString(XMLPublicKey);
```

***Exporting Private Keys**: As a rule of thumb, private keys should not be transmitted. They should remain in persistent storage and accessed only for decryption or digital signatures. When you export private keys, an alarm should be going off in your head to make sure they are not accidentally transmitted.*

Storing and Accessing Keys with CspParameters

Crypto service provider (CSP) objects like **RSACryptoServiceProvider** and **DSACryptoServiceProvider** (covered next chapter) require access to specific key pairs that represent the user or the application. **CspParameters** gives developers an easy way to store keys in the CryptoAPI key store. A string name or Handle can be used to store or retrieve keys from the key store through a **CspParameters** object. In order for **RSACryptoServiceProvider** or **DSACryptoServiceProvider** objects to access the keys, a **CspParameters** instance must be supplied in the class constructor.

***CspParameters Performance:** The performance benefit from using CspParameters can be noticeable in solutions that rely on the RSA class heavily, such as servers handling medium to high traffic loads that would otherwise generate a key pair every time an instance of RSACryptoServiceProvider is constructed. Solutions like this can see savings in terms of CPU* *usage that can translate to bigger savings on billing if the solution is hosted on the cloud where heavy CPU utilization can be expensive.*

Persisting keys with a **CspParameters** object is much different than using the three methods discussed above. **CspParameters** does not contain public import or export methods to handle keys. If the **CspParameters** object already has keys stored in the key store, the consuming object (RSA or DSA) will not generate a new key pair during construction. Below, a **CspParameters** object is created and its **KeyContainerName** is set to “userKeyName”. If there is no key container in the key store with the name of “userKeyName”, the **RSACryptoServiceProvider** will create a new key pair and it will be stored in a container and referenced by the **KeyContainerName**.

```
CspParameters csp = new CspParameters();
csp.KeyContainerName = "userKeyName";

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);
```

If the above code is executed repeatedly, the **RSACryptoServiceProvider** will not overwrite the key pair in the key container/key store. Instead, each time it is called it will use the first key pair that was generated and stored under the **KeyContainerName**. By default, keys are persisted to the user profile key store. The static **RSACryptoServiceProvider.UserMachineKeyStore** property is a read-only bool that indicates the key source: true for the machine key store, false for the user profile key store.

```
bool isMachineKeyStore = RSACryptoServiceProvider.UseMachineKeyStore;
```

RSACryptoServiceProvider can still import and export keys in a **CspBlob**, **RSAParameters**, and **XMLString** format while using a **CspParameters** object. Exports will merely export whatever key pair the **CspParameters** object is referencing. The caveat is that an import of public and private keys will overwrite the key pair in the container/key store. Here is an example of how this works:

```
CspParameters csp = new CspParameters();
csp.KeyContainerName = "userKeyName";

//The rsa instance gets its keys from the csp instance.
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);

//Another instance of RSA, rsa2, creates a new key pair.
RSACryptoServiceProvider rsa2 = new RSACryptoServiceProvider();

//The rsa2 instance exports its parameters that include the private key.
RSAParameters rsa2Params = rsa2.ExportParameters(true);

//The original rsa instance will import the rsa2 parameters,
//replacing its keys with the keys from rsa2 in the csp container.
rsa.ImportParameters(rsa2Params);
```

However, if only public keys are imported into the original instance, the instance will hold them locally and be able to use them, but will not persist them to key store and therefore the data contained in the **CspParameters** object will not be overwritten:

```
CspParameters csp = new CspParameters();
csp.KeyContainerName = "userKeyName";

```

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);

RSACryptoServiceProvider rsa2 = new RSACryptoServiceProvider();

RSAParameters rsa2Params = rsa2.ExportParameters(false);

rsa.ImportParameters(rsa2Params);

//rsa now contains the rsa2Params public key info, but the csp //container keys have not
been replaced.
```

__CryptoKeySecurity:__ .NET code will execute with full trust unless otherwise restricted. As a result, applications could have access to a CryptoAPI key store. __CryptoKeySecurity__ provides the ability to control access to a cryptographic key object without direct manipulation of an access control list (ACL). Every instance of __CspParameters__ contains an instance of __CryptoKeySecurity__ which can be adjusted to control permissions to that key in the key store. Developers should take advantage of this functionality and additional security through CAS (for legacy .NET) and ACLs. While these topics are more related to general .NET security administration and are outside the scope of this book, we want to mention them as they can pertain to secure key storage.

Using CspProviderFlags

The **CspProviderFlags** enum is used to modify the behavior of a **CspParameters** object. **CspProviderFlags** are accessible through the **CspParameters.Flags** property. Table 24 explains the behaviors associated with each.

Table 24: Description of CspProviderFlags Elements

CspProviderFlags Element	Description
NoFlags	Don't specify any settings.
NoPrompt	Prevent the CSP from displaying any user interface (UI) for this context.
UseArchivableKey	Allow a key to be exported for archival or recovery.
UseDefaultKeyContainer	Use key information from the default key container.
UseExistingKey	Use key information from the current key.
UseMachineKeyStore	Use key information from the computer's key store.
UseNonExportableKey	Use key information that cannot be exported.

UseUserProtectedKey	Notify the user through a dialog box or another method when certain actions are attempting to use a key. This flag is not compatible with the **System.Security.Cryptography.CspProviderFlags.NoPrompt flag**.

Deleting a Key from a CspParameters Container

You've seen how easy it is to create and store a key in a **CspParameters** object. Fortunately, deleting your key is just as easy. To delete a key you first need to create an instance of the **CspParameters** container and set the **KeyContainerName** property to the name of your key.

```
CspParameters keyContainer = new CspParameters();
keyContainer.KeyContainerName = "YourKeyName";
```

Next, create an instance of **RSACryptoServiceProvider** (this process will also work with **DSACryptoServiceProvider**) and pass the **CspParameters** object into the constructor. Set the **PersistKeyInCsp** property to **false.** Invoking the **Clear()** method will clear the container:

```
RSACryptoServiceProvider rsaCsp = new RSACryptoServiceProvider(keyContainer);
rsaCsp.PersistKeyInCsp = false;
rsaCsp.Clear();
```

Example: Two-Party Encryption and Decryption

The example above is intended to show the simplicity and the functionality of the encryption and decryption functions and does not deal with importing or exporting keys. Keys will need to be imported and exported the majority of the time when using RSA. The usual scenario involves Alice exporting and sending her public key to Bob. Bob will need to import Alice's key into an instance of **RSACryptoServiceProvider** and use the **Encrypt** method to be able to send her encrypted data, which she can decrypt with her private key.

Encrypting Bulk Data: *Do not use RSA for bulk data encryption. The two main reasons are length restriction (public key size and padding limit the maximum length of data that can be encrypted in a single operation) and performance (RSA is very slow compared to modern symmetric algorithms). For bulk data encryption, a symmetric algorithm should be used to protect the data and RSA can be used to protect the symmetric key. This will be covered shortly.*

For secured communication over a network, most protocols only use RSA encryption to provide secure key exchange for a symmetric key, and then switch to symmetric encryption once the symmetric key has been negotiated.

The first method imports a key blob object and encrypts data using this public key:

```
byte[] EncryptWithTheirPublicKey(byte[] data, byte[] publicKeyBlob)
{
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    rsa.ImportCspBlob(publicKeyBlob);
    return rsa.Encrypt(data, true);
}
```

Bob might call this function as follows:

```
//Obtain Alice's public key blob.
byte[] alicePublicKeyBlob = ...

byte[] plaintext = Encoding.UTF8.GetBytes("Dinner tonight at my place?");

byte[] ciphertext = EncryptWithTheirPublicKey(plaintext,alicePublicKeyBlob);


//The ciphertext can now be transmitted to Alice.
```

Alice receives the message and now has to decrypt it with her private key. Alice's decryption function will need to access her private key info and import it into an RSA object. Below, Alice retrieves her private key info from a secure file as XML format, uses **FromXMLString** to import the keys, and then decrypts:

```
byte[] DecryptWithMyPrivateKey(byte[] data)
{
    string xmlKeyInfo = File.ReadAllText("C:/.../secured file with ACLs");

    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    rsa.FromXmlString(xmlKeyInfo);

    return rsa.Decrypt(data, true);
}
```

Now Alice can use her decryption function to view Bob's message, and then get ready for dinner.

```
//Receive ciphertext from Bob
byte[] ciphertext=...

byte[] plaintext = DecryptWithMyPrivateKey(ciphertext);

string message = Encoding.UTF8.GetString(plaintext);
```

Keep in mind, Alice has no way of proving that this message is actually from Bob. However, if Bob and Alice were using Digital Signatures to provide nonrepudiation and integrity, they could ensure they were talking to each other, not Eve. Digital signatures are a crucial part of secure communications in public key channels and are covered next chapter.

RSACng

RSACng inherits from the **RSA** base class and provides similar functionality to the **RSACryptoServiceProvider** class, with exception to the provider and key store. One big difference between the **CryptoServiceProvider** and CNG providers for RSA is the default key size. **RSACng** will create a random 2,048-bit key upon construction unless a key is specified; whereas **RSACryptoServiceProvider** will default to a 1,024-bit key. This mostly reflects the difference in time between when the two classes were implemented. **RSACryptoServiceProvider** has been around since .NET 1, when 1,024-bit keys were still an industry standard and considered secure. Conversely,

RSACng made a much later entrance in .NET 4.6 where 2,048-bit keys are the minimum acceptable length and 1,024-bit keys are considered insecure for most purposes.

Creating an Instance

An instance of **RSACng** can be created easily with the **new** keyword. Each new instance will be instantiated with a randomly generated 2,048-bit key pair.

```
RSACng rsaCng = new RSACng();
```

Remember, using **RSA.Create()** creates an instance of **RSACryptoServiceProvider** (this could change in future versions of .NET).

Encryption and Decryption

RSACng inherits from **RSA**, and its **Encrypt** and **Decrypt** methods will work the same as the **RSA** base class example. Unlike **RSACryptoServiceProvider**, **RSACng** cannot specify padding using a bool value; it must use **RSAEncryptionPadding**. Below, an instance of **RSACng** will use a randomly generated key pair to encrypt and decrypt byte-array data:

```
RSACng rsa = new RSACng();

byte[] data = new byte[32];

byte[] ciphertext = rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA1);

byte[] plaintext = rsa.Decrypt(ciphertext, RSAEncryptionPadding.OaepSHA1);
```

Working with Cryptography Next Generation (CNG) Keys

The **CngKey** object provides means to create, store, format, and open keys to use with different (Cryptography Next Generation) CNG algorithms. Its purpose is similar to the **CspParameters** objects we covered earlier this chapter, but **CngKey** provides a different interface and functionality. In this section we use the **RSACng** and **ECDiffieHellmanCng** algorithms to exemplify the **CngKey** functionality (**ECDiffieHellmanCng** will be covered later in the chapter). This section will also teach you how to handle the keys that are used with the **ECDsa** algorithm in chapter 9.

Quick Start for RSACng

Before we go through how **CngKey** objects are used, we'll provide a quick start example for **RSACng** that creates and stores a **CngKey** (named/referenced as "myRsaCngKey") in the key store. Once the key object has been obtained by being opened or created, it is supplied to the **RSACng** instance through the class constructor:

```
string keyName = "myRsaCngKey";

CngKey key;

if (CngKey.Exists(keyName))
{
    key = CngKey.Open(keyName);
}
else
{
    key = CngKey.Create(CngAlgorithm.Rsa, keyName);
}
```

```
RSACng rsaCng = new RSACng(key);
```

Choosing a CNG Algorithm

Before we create a new **CngKey**, we need to know what algorithm we will be using it with. The **CngAlgorithm** class encapsulates CNG implementations of popular algorithms as static read-only properties, each of which will return a **CngAlgorithm** object.

- ECDiffieHellmanP256
- ECDiffieHellmanP384
- ECDiffieHellmanP521
- ECDsaP256
- ECDsaP384
- ECDsaP521
- MD5
- Rsa
- Sha1
- Sha256
- Sha384
- Sha512

Creating and Opening Keys

To create a key and add it to the key store, a **CngAlgorithm** and a key name (used for a look up) must be supplied as arguments to the static **Cng.Create** method. This key would be used by an instance of the **RSACng** class:

```
CngKey key = CngKey.Create(CngAlgorithm.Rsa, "myRsaCngKey");
```

Ephemeral keys that do not require key store interaction can be created without a name argument:

```
CngKey key = CngKey.Create(CngAlgorithm.Rsa);
```

But this is often unnecessary because algorithms instances will usually generate their own random key pair if one is not provided.

The static **CngKey.Open** method is used to open an existing key. Different overloads are available to reference the key, including the key name, a **Microsoft.Win32.SafeHandles.SafeNCryptKeyHandle**, and a **CngProvider** to reference a specific key store. Below “myRsaCngKey” (created above) is opened:

```
CngKey key = CngKey.Open("myRsaCngKey");
```

An error will be thrown if you try to create a key that already exists or if you try to open a key that doesn’t exist. The immediate issue that arises when programming with **CngKey** objects is that keys must be *created* the first time the code is executed, but *opened* every time thereafter. Fortunately, the **CngKey** class saw this coming and included an **Exists** method to deal with this issue. The following code will open a key if it exists and create one if it doesn’t:

```
string keyName = "myRsaCngKey";
```

```
CngKey key;

if (CngKey.Exists(keyName))
{
    key = CngKey.Open(keyName);
}
else
{
    key = CngKey.Create(CngAlgorithm.Rsa, keyName);
}
```

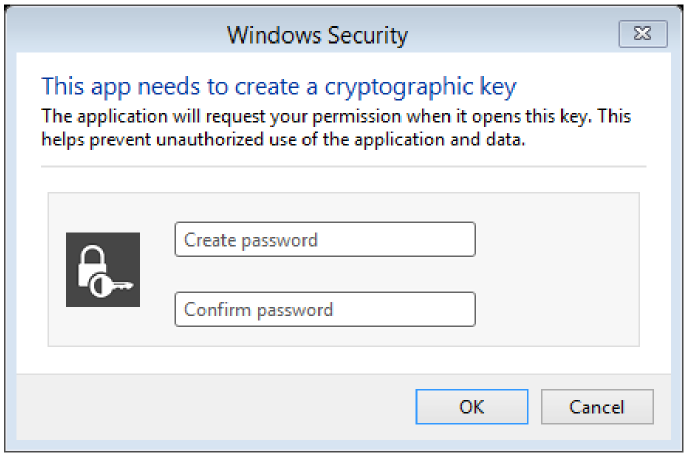
Setting Key Creation Parameters

A **CngKeyCreationParameters** object can be created and supplied in the static **CngKey.Create** method during key creation. This is used to control the behavior and usage of a key. Export policies, key usages, and UI policies are commonly adjusted. Below, an instance of the **CngKeyCreationParameters** class is created and set to allow exports, all key usages, and a **CngUIPolicy** that forces a prompt for the user to create a password for key access:

```
CngKeyCreationParameters cngParams = new CngKeyCreationParameters();
cngParams.ExportPolicy = CngExportPolicies.AllowExport;
cngParams.KeyUsage = CngKeyUsages.AllUsages;

CngUIPolicy ui = new CngUIPolicy(CngUIProtectionLevels.ForceHighProtection);

cngParams.UIPolicy = ui;

CngKey key = CngKey.Create(CngAlgorithm.Rsa, keyName,cngParams);
```

When the code is ran for the first time the UI prompt will appear for the key access, as set in the **CngUIPolicy**. Figure 33 shows a prompt generated from a **CngUIPolicy.**

Figure 33: CngUIPolicy Prompt



This app needs to create a cryptographic key

The application will request your permission when it opens this key. This helps prevent unauthorized use of the application and data.

You may not always need to set key creation parameters for your keys. However, in some cases it will be the only way to achieve the intended behavior or usage.

Importing and Exporting Keys

Existing key material can be imported into a **CngKey** object using the static **CngKey.Import** method. This method takes a byte array key blob and formats it into a key according to the **CngKeyBlobFormat** object specified in the parameters. Exporting is performed by calling the **Export** method of a **CngKey** instance and specifying a **CngKeyBlobFormat** as a parameter.

CngKeyBlobFormat contains the following formats and exposes them as static read-only properties. By selecting a property you are therefore selecting the Type corresponding to the property name:

- EccPrivateBlob
- EccPublicBlob
- GenericPrivateBlob
- GenericPublicBlob
- OpaqueTransportBlob
- Pkcs8PrivateBlob

Most situations call for either generic or ECC blob types. Keep in mind that keys must be imported and exported as the same type to format correctly.

The **CngKey** class only exports keys as byte arrays, but formatting options are still adjustable with **CngKeyBlobFormat.** Classes like **ECDsaCng** and **ECDiffieHellmanCng** have better variety in their packaging and can export their object (including the key) as an XML string.

An **RSACng** key is exported below using the **GenericPublicBlob** format:

```
byte[] publicBlob = key.Export(CngKeyBlobFormat.GenericPublicBlob);
```

The static **Import** method is used to import keys by specifying the **CngKeyBlobFormat**. **GenericPublicBlob** is used below:

```
CngKey publicKey = CngKey.Import(publicBlob, CngKeyBlobFormat.GenericPublicBlob);
```

Using a Key with an Algorithm

Once you have an instance of a **CngKey** object, it can be supplied to CNG algorithms through their constructor:

```
RSACng rsaCng = new RSACng(key);
```

Here's a more cumulative example of what we've covered this section:

```
CngKeyCreationParameters cngParams = new CngKeyCreationParameters();
cngParams.ExportPolicy = CngExportPolicies.AllowExport;
cngParams.KeyUsage = CngKeyUsages.AllUsages;

CngUIPolicy ui = new CngUIPolicy(CngUIProtectionLevels.ForceHighProtection);

cngParams.UIPolicy = ui;

string keyName = "myRsaCngKey";

CngKey key;

if (CngKey.Exists(keyName))
{
    key = CngKey.Open(keyName);
}
else
{
    key = CngKey.Create(CngAlgorithm.Rsa, keyName, cngParams);
}

RSACng rsaCng = new RSACng(key);
```

Below, we'll create an **ECDiffieHellmanP521** key for use with the **ECDiffieHellmanCng** algorithm, which will be covered later this chapter.

```
CngKey dh521Key = CngKey.Create(CngAlgorithm.ECDiffieHellmanP521, "yourDH521Key");

ECDiffieHellmanCng ecDhCng = new ECDiffieHellmanCng(dh521Key);
```

Example: Sharing RSAParameters between RSA Classes

RSACng can also use the **RSAParameters** object to handle RSA key material. **RSA** subclasses can be used to import and export **RSAParameters** objects. The code below shows an **RSACryptoServiceProvider** object exporting **RSAParameters** which are then imported by an **RSACng** instance:

```
RSACryptoServiceProvider rsaCsp = new RSACryptoServiceProvider();
var publicRsa = rsaCsp.ExportParameters(false);

RSACng rsaCng = new RSACng();
rsaCng.ImportParameters(publicRsa);
```

Hybrid Encryption

RSA is not suitable for performing bulk data encryption. Assuming that you have an RSA key pair and need to protect bulk data, the most secure and efficient pattern is to randomly generate a symmetric key, encrypt the data using a symmetric algorithm such as AES or Rijndael, and then protect the symmetric key with RSA. The encrypted symmetric key is attached to the encrypted data because it is not in plaintext and is required for decryption. Upon decryption, the encrypted symmetric key is removed from the ciphertext, decrypted with RSA, and then used to decrypt the ciphertext. A database or other type of lookup could be used to store the encrypted symmetric key as well; however, keeping it with the data is simple.

Example: RSA-AES Hybrid Encryption

Below, two methods are used to encrypt and decrypt bulk data using **RSACryptoServiceProvider** and a symmetric algorithm. This first method, *Rsa_AesEncrypt*, randomly generates a symmetric key and passes the key along with the plaintext data to a method called *SymmetricEncrypt*. This method is assumed to function the same as our AES encryption and decryption examples from page 89. Finally, the symmetric key is encrypted with an instance of **RSACryptoServiceProvider** and prepended to the ciphertext:

```
public byte[] Rsa_AesEncrypt(RSA rsa, RSAEncryptionPadding padding, byte[] plaintext)
{
    byte[] symmkey = new byte[32];

    using (RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider())
    {
        rng.GetBytes(symmkey);
    }

    byte[] ciphertext = SymmetricEncrypt(plaintext, symmkey);

    byte[] encryptedSymmKey = rsa.Encrypt(symmkey, padding);

    return encryptedSymmKey.Concat(ciphertext).ToArray();
}
```

Next, decryption takes place by removing the protected symmetric key from the ciphertext and decrypting it with the RSA private key. For clarity, this RSA private key should correspond to the public key that was used to perform the encryption. Once the symmetric key has been decrypted, we can decrypt the data and return the plaintext. The *SymmetricDecrypt* method is assumed to decrypt a ciphertext created by the *SymmetricEncrypt* method above (for an example on how these methods work look at the simple AES examples on page 89):

```
public byte[] Rsa_AesDecrypt(RSA rsa, RSAEncryptionPadding padding, byte[] ciphertext)
{
    byte[] encryptedSymmKey = ciphertext.Take(rsa.KeySize >> 3).ToArray();

    byte[] rawCiphertext = ciphertext.Skip(rsa.KeySize >> 3).ToArray();

    byte[] symmKey = rsa.Decrypt(encryptedSymmKey, padding);

    return SymmetricDecrypt(rawCiphertext, symmKey);
}
```

The methods can be easily tested with a randomly generated RSA key pair. Production environments should have secured a key pair that is retrievable in similar style.

```
byte[] data = new byte[32];

RSACng rsa = new RSACng();

byte[] ciphertext = Rsa_AesEncrypt(rsa, RSAEncryptionPadding.OaepSHA256, data);

byte[] plaintext = Rsa_AesDecrypt(rsa, RSAEncryptionPadding.OaepSHA256, ciphertext);
```

RSA Key Exchange

RSA encryption isn't used very often as a standalone cipher to maintain an entire secure session between parties. Nor should it be. Instead, RSA is used to securely exchange a key which will perform bulk encryption and decryption through a symmetric algorithm like AES. This is mostly for performance reasons; symmetric ciphers are much faster. It's also because the amount of data that RSA can encrypt is limited by its key size and padding, rendering RSA impractical as a primary or bulk data cipher.

In this chapter we've covered the encryption and decryption methods in the **RSACryptoServiceProvider** class. It's easy enough to use the simple RSA examples we've shown to encrypt and decrypt a symmetric key, but .NET also provides classes geared toward RSA-based key exchange. Under best practices, these should be used for the purposes of exchanging symmetric keys instead of using **RSACryptoServiceProvider** directly.

These classes include:

- RSAOAEPKeyFormatter
- RSAOAEPKeyDeformatter
- RSAPKCS1KeyFormatter
- RSAPKCS1KeyDeformatter

The formatter variations perform the secure formatting/encryption (**RSAOAEPKeyFormatter** and **RSAPKCS1KeyFormatter**). Likewise, deformatting/decryption is performed by the deformatter variation (**RSAOAEPKeyDeformatter** and **RSAPKCS1KeyDeformatter**). Padding is decided at the class level, hence the RSAPKCS1… and RSAOAEP… class names, which use PKCS#1 or OAEP padding (padding was covered in the **RSACryptoServiceProvider** section). Last section we discussed the preference of OAEP padding to PKCS#1. This recommendation will also apply to this section with the **RSAOAEPKeyFormatter** and **RSAOAEPKeyDeformatter** classes. Since the OAEP and PKCS1 variations share an identical interface, we will only show the use of OAEP.

RSAOAEPKeyFormatter and RSAOAEPKeyDeformatter

Instances of **RSAOAEPKeyFormatter** and **RSAOAEPKeyDeformatter** can be created with either a parameterless constructor, or by supplying an instance of **AsymmetricAlgorithm** in the constructor. The instance of **AsymmetricAlgorithm** must contain a key.

RSAOAEPKey[De]Formatter Compatibility: *At the time of this writing (.NET 4.6) the* ***RSACryptoServiceProvider*** *class is the only concrete* ***AsymmetricAlgorithm*** *in .NET that will work with the* ***RSAOAEPKey[De]Formatter*** *classes.*

We've already covered how to import keys with **RSACryptoServiceProvider**. **RSACryptoServiceProvider** inherits **AsymmetricAlgorithm**, which means we can use it to provide a public key to an **RSAOAEPKeyFormatter** or a private key to an **RSAOAEPKeyDeformatter** object. Below, an instance of **RSACryptoServiceProvider** imports a key and then is passed to the constructor of a new **RSAOAEPKeyExchangeFormatter** object:

```
byte[] publicKeyBlob =...

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
rsa.ImportCspBlob(publicKeyBlob);

RSAOAEPKeyExchangeFormatter fmt = new RSAOAEPKeyExchangeFormatter(rsa);
```

Setting Keys in RSAOAEPKeyFormatter or RSAOAEPKeyDeformatter: *If an instance is not supplied through the constructor, the* ***AsymmetricAlgorithm*** *instance will have to be provided using the* ***SetKey*** *method.*

The formatter is now ready to encrypt a symmetric key. This is done using the **RSAOAEPKeyExchangeFormatter.CreateKeyExchange** method. Here the code uses the formatter instance (fmt) from above:

```
byte[] privateSymmKey = ...

byte[] keyEx = fmt.CreateKeyExchange(privateSymmKey);
```

The byte array *keyEx* contains the encrypted symmetric key that is sent to the recipient. Upon receiving the encrypted key, decryption is performed with the deformatter class using the recipient's private key.

Unlike direct encryption using **RSACryptoServiceProvider** where encryption and decryption take place within the same class, key exchange formatters require two separate objects. Decryption will always be performed with the "deformatter" variation. Instances are created comparatively with the formatter variation we used above. Here we will import the private key from a CryptoAPI key store. However, the key could be imported using any of the methods covered last section.

```
CspParameters csp= new CspParameters();
csp.KeyContainerName="someUserKeyPair";

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);

RSAOAEPKeyExchangeDeformatter deFmt = new RSAOAEPKeyExchangeDeformatter(rsa);
```

Once the encrypted symmetric key is received from the sender it can be decrypted using the **RSAOAEPKeyExchangeDeformatter.DecryptKeyExchange** method:

```
byte[] keyEx =...

byte[] symmKey = deFmt.DecryptKeyExchange(keyEx);
```

Turn to SSL/TLS before Trying to Write Your Own Secure Session Protocol: *In a session-type RSA model, communication is initiated and RSA is used to negotiate (exchange) a symmetric key. In most cases, all further communications in the session rely on the symmetric key.*

For application level solutions, you can get this type of behavior by using SSL/TLS to secure the communications between two endpoints. This is far easier and almost always more secure than trying to write your own session protocol using RSA (or another asymmetric algorithm). There are lots of resources that explain how to enable SSL/TLS between a client and server and enhance trust through the use of a certificate authority (CA).

Example: Key Exchange with Symmetric Encryption

This example shows how the RSA key formatter and deformatter classes can be used with the simple AES methods on page 89. A single RSA key pair will be created to illustrate encrypting a symmetric key with an RSA public key and decrypting with the corresponding RSA private key. As you may notice, this example does not maintain any type of session between the parties after the symmetric key is exchanged. Instead, it only allows for bulk encryption between them where the RSA key exchange formatter and deformatter objects must encrypt and decrypt a symmetric key for every message. Next, we'll show you how the encryption and decryption methods work.

This first method, *FormatRsaToAesEncrypt*, handles encryption. A random symmetric key is generated and passed to an AES encryption method (any secure symmetric algorithm will also work) and the symmetric key is encrypted using **RSAOAEPKeyFormatter** (remember, this object is instantiated using the **RSA** instance in the method parameters) and the encrypted key and the AES ciphertext are concatenated and returned.

```
static byte[] FormatRsaToAesEncrypt(RSA rsa, byte[] input)
{
    byte[] symmKey = new byte[32];

    new RNGCryptoServiceProvider().GetBytes(symmKey);

    byte[] encryptedData = AesEncrypt(input, symmKey);

    RSAOAEPKeyExchangeFormatter fmt = new RSAOAEPKeyExchangeFormatter(rsa);
    byte[] keyex = fmt.CreateKeyExchange(symmKey);

    byte[] ciphertext = new byte[keyex.Length + encryptedData.Length];

    Buffer.BlockCopy(keyex, 0, ciphertext, 0, keyex.Length);
    Buffer.BlockCopy(encryptedData, 0, ciphertext, keyex.Length, encryptedData.Length);

    return ciphertext;
}
```

This next method, *DeformatRsaToAesDecrypt,* handles decryption. The encrypted symmetric key is removed from the input array and subsequently decrypted using the **RSAOAEPKeyExchangeDeformatter** class (remember, this object is instantiated using the **RSA** instance in the method parameters). The encrypted data and the recovered symmetric key are passed to the AES decryption method and the plaintext is returned.

```
static byte[] DeformatRsaToAesDecrypt(RSA rsa, byte[] input)
{
    byte[] keyex = new byte[rsa.KeySize >> 3];
    Buffer.BlockCopy(input, 0, keyex, 0, keyex.Length);

    RSAOAEPKeyExchangeDeformatter fmt = new RSAOAEPKeyExchangeDeformatter(rsa);
    byte[] symmkey = fmt.DecryptKeyExchange(keyex);

    return AesDecrypt(input.Skip(keyex.Length).ToArray(), symmkey);
}
```

We assume in both of the above methods that symmetric encryption and decryption are being performed in the *AesEncrypt* and *AesDecrypt* calls.

Elliptic Curve Diffie-Hellman Key Agreement

Diffie-Hellman (DH) is another algorithm for securely distributing a public key between two parties. Conceptually, DH key exchange is different from RSA. If you recall, the normal operation of RSA requires a party to come up with a private symmetric key and give it to the other party; at this point you have completed the RSA *key exchange*. Conversely, DH does not require a party to produce a secret symmetric key to give to the other party. This concept works by using the parties' asymmetric key pairs to create a symmetric key that is unique to the combination of their keys; different key combinations should not (with very low probability) produce the same symmetric key. This is more specifically referred to as the DH *key agreement*, but some people still refer to it generally as key exchange. As an example of how this agreement takes place, Bob and Alice would both have a public and private Diffie-Hellman key pair, they would exchange public keys and use the DH algorithm to compute a symmetric key based off of their own private key and the other party's public key. Alice and Bob will both produce the same symmetric key through this *key agreement*. Figure 34 illustrates how Bob uses Alice's public key to generate a symmetric key.

Figure 34: Bob uses Alice's public key to generate a symmetric key

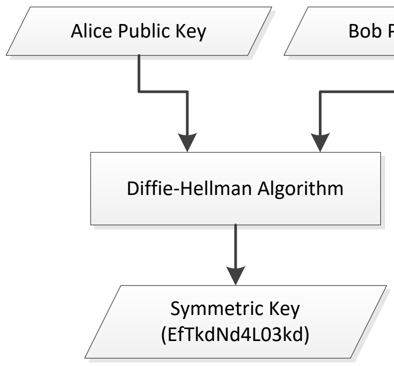
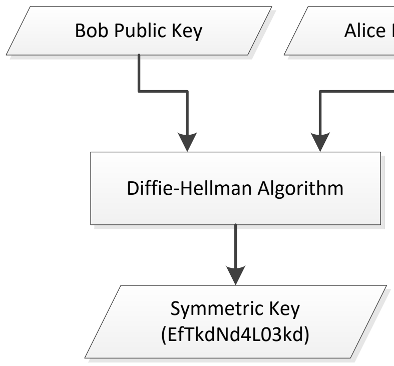


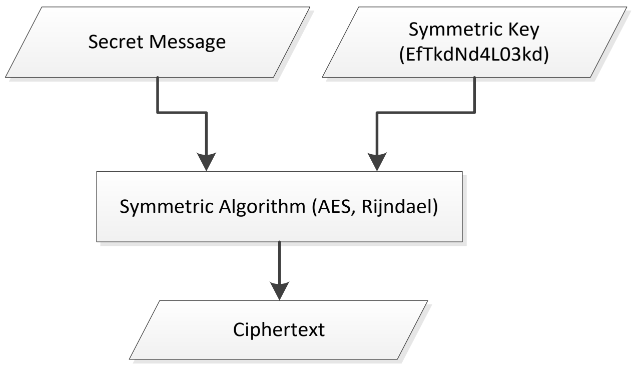
Figure 35 illustrates how Alice uses Bob's public key to produce the same symmetric key as Bob.

Figure 35: Alice uses Bob's public key to produce the same symmetric key as Bob



Once the key agreement is complete, encryption can be performed with the shared symmetric key. Figure 36 shows a simple encryption process used by Alice and Bob (with the same shared symmetric key). Here, the same key would also be used for decryption.

Figure 36: Alice and Bob both possess the same symmetric key to encrypt



ECDiffieHellmanCng

ECDiffieHellman is the abstract base class for all implementations of Elliptic Curve Diffie-Hellman in .NET. The only concrete implementation is **ECDiffieHellmanCng** as of this writing. You caught a glimpse of **ECDiffieHellmanCng** at the end of the **CngKey** section. Here we'll learn how to perform secure symmetric key agreements between parties and how to handle key material in different formats.

Creating an instance is relatively easy. **ECDiffieHellmanCng** uses a randomly generated key pair if a **CngKey** object is not used in the constructor:

```
ECDiffieHellmanCng ecDh = new ECDiffieHellmanCng();
```

If a **CngKey** is being used from the key store, it must be opened (and exist) before the **ECDiffieHellmanCng** instance consumes it:

```
if (!CngKey.Exists("dh521Key"))
{
    CngKey.Create(CngAlgorithm.ECDiffieHellmanP521, "dh521Key");
}

CngKey dhKey = CngKey.Open("dh521Key");

ECDiffieHellmanCng dh = new ECDiffieHellmanCng(dhKey);
```

The primary functionality of the **ECDiffieHellmanCng** class is built around the **DeriveKeyMaterial** method. This method produces a symmetric key by using the private key from one party and the public key from another. The private key is held by the **ECDiffieHellmanCng** instance and the public key (**CngKey** or **ECDiffieHellmanPublicKey**) is supplied through the parameter of **DeriveKeyMaterial**. Below, an **ECDiffieHellmanCng** instance is created using what is assumed to be existing private key material. Another party's public key material is then imported into a **CngKey** (the imported byte array key is assumed to be in **GenericPublicBlob** format). This **CngKey** is then passed to the **ECDiffieHellmanCng** instance's **DeriveKeyMaterial** method, which returns a byte array of symmetric key material.

```
CngKey privateKeyInfo = ...

ECDiffieHellmanCng dh = new ECDiffieHellmanCng(privateKeyInfo);

byte[] publicKeyBlob;

CngKey importedKey = CngKey.Import(publicKeyBlob,CngKeyBlobFormat.GenericPublicBlob);

byte[] symmetricKey = dh.DeriveKeyMaterial(importedKey);
```

The same symmetric key will be produced if the other party repeats this process on their end.

The **ECDiffieHellmanCng** class also offers the ability to use a shared secret in the key agreement process. This could increase security since an attacker would have to produce the shared secret in addition to one of the party's private keys. However, both parties would have to possess the secret data before the agreement is made. This introduces a chicken-or-the-egg type of problem. The point of DH is to securely provide both parties with shared symmetric keys across an insecure public channel. So how would you securely negotiate the shared secret without already having a secure channel? The .NET implementations just assume both parties have the shared secret and leaves it up to the developer. In most cases the shared secret will be negotiated through an out-of-band channel like a separate secure protocol/network, or even as simple as a face-to-face communication. Ultimately, we have to make the same assumption that you have already negotiated the secret. Now, making this assumption, let’s look at using a shared secret.

The **SecretAppend** or **SecretPrepend** properties can be used in an **ECDiffieHellmanCng** instance to introduce a shared secret into the key agreement process. Both parties must use the same shared secret and specify the correct method (append or prepend). Here we will append a byte array that is assumed to contain the shared secret to the DH key agreement:

```
byte[] sharedSecret=...

ECDiffieHellmanCng ecDh = new ECDiffieHellmanCng();
ecDh.SecretAppend = sharedSecret;
byte[] symmKey = ecDh.DeriveKeyMaterial(...);
```

Besides byte array key information imported and exported through **CngKey** objects, algorithm-specific key material can also be imported and exported using XML strings through an **ECDiffieHellmanCng** instance. The **FromXMLString** and **ToXMLString** methods import and export (respectively) XML formatted string data. Both methods require a parameter to specify the type of formatting. Formats can be selected from the **ECKeyXMLFormat** enum, but **ECKeyXMLFormat.Rfc4050** is currently the only implemented format.

Exporting to XML:

```
ECDiffieHellmanCng ec = new ECDiffieHellmanCng();
string xml = ec.ToXmlString(ECKeyXmlFormat.Rfc4050);
```

Importing from XML:

```
string xml;
ECDiffieHellmanCng ec = new ECDiffieHellmanCng();
ec.FromXmlString(xml, ECKeyXmlFormat.Rfc4050);
```

Below, a **P256** and a **P521 ECDiffieHellmanCng** key are shown after being exported as XML. You can see that the primary difference between the two is the length of the “X Value” and “Y Value” fields.

P256 XML

```
<ECDHKeyValue xmlns=\"http://www.w3.org/2001/04/xmldsig-more#\">\r\n
<DomainParameters>\r\n    <NamedCurve URN=\"urn:oid:1.2.840.10045.3.1.7\" />\r\n
</DomainParameters>\r\n  <PublicKey>\r\n    <X
Value=\"35788933255588561950116519884391738395218385320420100996787227654715592432495\"
xsi:type=\"PrimeFieldElemType\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
/>\r\n    <Y
Value=\"45169012077394825836638093128459440013953841306949441451189372310126483596 58\"
xsi:type=\"PrimeFieldElemType\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
/>\r\n  </PublicKey>\r\n</ECDHKeyValue>
```

P521 XML

```
<ECDHKeyValue xmlns=\"http://www.w3.org/2001/04/xmldsig-more#\">\r\n
<DomainParameters>\r\n    <NamedCurve URN=\"urn:oid:1.3.132.0.35\" />\r\n
</DomainParameters>\r\n  <PublicKey>\r\n    <X
Value=\"37518521872775063345730483191776485817030832357007110901792613108207662314523628914
3488078505524858552225729596487265263196133330714003140212410297310610877\"
xsi:type=\"PrimeFieldElemType\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
/>\r\n    <Y
Value=\"18630934330062487973470432112831948044513502188432144812260523249507687303041826337
66707738426958462565367073592189971482949253148868944794040368640236324637\"
xsi:type=\"PrimeFieldElemType\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
/>\r\n  </PublicKey>\r\n</ECDHKeyValue>
```

Key Agreement Example

We'll show a simple key agreement between Alice and Bob. The logical starting point is exporting a public key for each of the parties so it can be exchanged. Keep in mind that Alice and Bob must specify the same **CngAlgorithm** when creating their keys and they need to export and import in the same format. **CngKeyBlobFormat.GenericPublicBlow** is used here:

```
byte[] AliceExportKey()
{
    if (!CngKey.Exists("aliceKey"))
    {
        CngKey.Create(CngAlgorithm.ECDiffieHellmanP521, "aliceKey");
    }

    CngKey aliceKey = CngKey.Open("aliceKey");

    return aliceKey.Export(CngKeyBlobFormat.GenericPublicBlob);
}
```

```
byte[] BobExportKey()
{
    if (!CngKey.Exists("bobKey"))
    {
        CngKey.Create(CngAlgorithm.ECDiffieHellmanP521, "bobKey");
    }

    CngKey bobKey = CngKey.Open("bobKey");

    return bobKey.Export(CngKeyBlobFormat.GenericPublicBlob);
}
```

We assume that some type of reliable protocol such as TCP/IP is being used for the public channel to exchange Alice and Bob's public keys. So, assuming Bob has sent his key to Alice, and Alice has sent her key to Bob, we write the functions to produce the symmetric key for both parties. Each party goes through the following general steps:

1. Open their own private key in a **CngKey** object.
2. Create an instance of **ECDiffieHellmanCng** using their private key object.
3. Import the other party's key into a **CngKey** object.
4. Use the **ECDiffieHellmanCng** instance to derive a symmetric key using the party's imported key.

Alice uses these steps in the following method, passing in Bob's public key as the parameter.

```
byte[] GetSymmetricKeyAlice(byte[] publicKeyBlob)
{
    CngKey aliceKey = CngKey.Open("aliceKey");

    ECDiffieHellmanCng ecDh = new ECDiffieHellmanCng(aliceKey);

    CngKey bobKey = CngKey.Import(publicKeyBlob, CngKeyBlobFormat.GenericPublicBlob);

    return ecDh.DeriveKeyMaterial(bobKey);
}
```

Conversely, Bob uses his own version, passing in Alice's public key.

```
byte[] GetSymmetricKeyBob(byte[] publicKeyBlob)
{
    CngKey bobKey = CngKey.Open("bobKey");

    ECDiffieHellmanCng ecDh = new ECDiffieHellmanCng(bobKey);

    CngKey aliceKey = CngKey.Import(publicKeyBlob,CngKeyBlobFormat.GenericPublicBlob);

    return ecDh.DeriveKeyMaterial(aliceKey);
}
```

After the exchange is complete and Alice and Bob have derived their keys, they communicate using a symmetric algorithm like AES over the presumably insecure public channel. Remember that digital signatures should also be used to enforce nonrepudiation and integrity where these might be of concern (this is covered next chapter).

Chapter Summary

- Asymmetric encryption algorithms use a public key and a private key. Public keys are used for encryption and private for decryption.
- Asymmetric algorithms should not be used for bulk data encryption. Instead, asymmetric algorithms are used to securely negotiate a symmetric key, which is then used to perform bulk data encryption with a symmetric algorithm such as AES.
- Most algorithms have provisions to import and export keys in different formats. Be careful not to accidentally export and distribute your private key.
- **CngKey** and **CspParameters** objects coordinate access to user and machine key stores where keys can be persisted and accessed. Be careful; applications that execute with full trust and permissions can access key stores.
- Elliptic Curve (EC) algorithms, such as **ECDiffieHellman**, have comparable security to non-EC algorithms, but are typically more efficient. These are becoming more popular, especially in mobile solutions.

Chapter Questions and Exercises

1. Explain the process of using asymmetric and symmetric encryption (hybrid encryption) to secure bulk data. Why is this a common practice?

2. Describe the changes that occurred in the RSA base class in .NET 4.6. What are the differences between **RSACryptoServiceProvider** and **RSACng**? How are they similar? Which one will probably have to be used with legacy systems?
3. How is Diffie-Hellman key agreement different from an RSA key exchange? Describe each.
4. What are the minimum recommended key sizes for each of the asymmetric encryption and key exchange/agreement algorithms in .NET?
5. Which type of padding should be used for asymmetric encryption? What styles are available in .NET? Have these changed with .NET 4.6? If so, how?

Scenarios

1. You're working on a development team for an application that needs to perform data encryption on messages that are transmitted between users. Your boss suggests that RSA should be used as a mechanism to exchange symmetric keys between users, but asks if there are any other options that should be considered. What should you tell her? If there are other options, what might make them more or less secure? How might they impact the system?

9 Digital Signatures

Digital Signature: *The result of a cryptographic transformation of data that, when properly implemented, provides origin authentication, assurance of data integrity and signatory non-repudiation.*

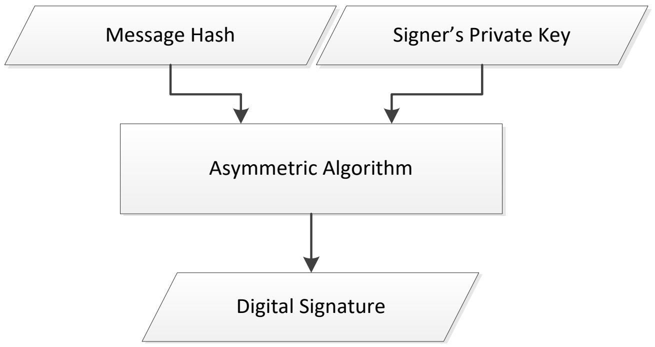
Chapter Objectives

1. Understand how digital signatures provide nonrepudiation.
2. Learn how digitally sign and verify data using .NET implementations of RSA, DSA, and EC-DSA.
3. Explain the different orders of combining symmetric encryption and digital signing, and their benefits and drawbacks.
4. Describe the implications of the Horton principle with digital signatures.

Digital signatures solve one of the other problems with symmetric key systems: Traditional symmetric key MACs cannot offer nonrepudiation. *Nonrepudiation* means that a party is unable to deny having performed an action. With a symmetric key MAC tc7here is no way of knowing which key holder encrypted and subsequently authenticated the message. Therefore, a party that generates a MAC tag for a message is not directly linked to the message through any type of identity; a receiver would have no way of determining if the correct key holder created the message—or which key holder created the message. In a public key system, each entity has its own public/private key pair that is linked to it and only it. Assuming that trust has been established (typically through a certificate authority (CA) or local trust list), this facet of asymmetric encryption—digital signing—is what can provide nonrepudiation.

A digital signature is created in an opposing fashion to how an encrypted (asymmetric) message is encrypted. Asymmetric encryption works on the basis of encrypting data with the receiver's public key, and the receiver decrypting with their private key. A digital signature is added by generating a cryptographic hash of the message and encrypting the hash with the sender's private key. Figure 37 shows a basic digital signature.

Figure 37: Basic Digital Signature



The receiver can decrypt the hash with the sender's public key and verify that a computed hash of the message matches the received hash. This verifies two things: message integrity, and the sender's identity associated with the key pair.

The two main algorithms for computing digital signatures are RSA and DSA. We covered RSA encryption last chapter; however, RSA can also be used for digital signatures. Digital Signature Algorithm (DSA) is implemented by .NET in the **DSACryptoServiceProvider** class. Elliptic Curve Cryptography (ECC) is also implemented in .NET with digital signatures through the **ECDsa** class (the EC prefix denotes "elliptic curve").

Specifying Hash Algorithms for Signing

For efficiency and simplicity, digital signatures are used to sign a hash rather than raw data itself. As a result, hash algorithms are commonly referenced and used with digital signing. Some of the signature methods of cryptographic objects in .NET need an Object Identifier (OID) that corresponds to a specific hash algorithm, or an instance of a particular algorithm; .NET 4.6 has added an object just to specify a hash algorithm in signature methods.

Older versions of .NET need an OID or string name supplied to specify the hash algorithm. Table 25 contains the OIDs for the MD5 and SHA algorithms (this may come in handy later if you're using older versions of .NET):

Table 25: OIDs for MD5 and SHA-2 series algorithms

Hash Algorithm	OID
MD5	1.2.840.113549.2.5
SHA1	1.3.14.3.2.26
SHA256	2.16.840.1.101.3.4.2.1
SHA384	2.16.840.1.101.3.4.2.2
SHA512	2.16.840.1.101.3.4.2.3

We are not including a table for string hash names because the strings are simply the same as the algorithm name, i.e. "SHA256", "SHA1".

Similarities in Signature Object Functionality

Most cryptographic objects in .NET that perform digital signing have very predictable behavior and usage because they share a few key methods. The first are signing methods: **SignData** and **SignHash**. These both compute a digital signature. **SignData** performs hashing internally and generates a signature from the hash. **SignHash** signs a hash value rather than raw data and places the responsibility of generating a hash on the developer. Most classes implement both methods. Depending on the algorithm and class, parameters will vary and may include overloads. Some classes for instance, only require the value (data or hash) to compute a signature. Others may need additional information such as the hash algorithm to use or the type of padding.

VerifyData and **VerifyHash** complement the signing methods. These methods verify the data that is assumed to have been signed using the **SignData** and **SignHash** methods, respectively. Like the signing methods, these are distinguished by whether they accept raw data or a hash value. Verification methods will need the purported message as well as the signature, and will require the same parameter specifications used to sign the data. For example, if the data is signed using **Pss** signature padding, **Pss** must be used during verification.

Digital Signing with RSA in .NET

The RSA Base Class

RSA digital signatures work with the same style of public/private key pair that is used for regular asymmetric encryption. Key material is still handled the same way as in the RSA section last chapter.

.NET 4.6

Starting in .NET 4.6 the **RSA** base class defines two signature methods: **SignData** and **SignHash**. **SignData** hashes data using a specified hash algorithm and then signs the data. **SignHash** signs an existing hash.

The corresponding **VerifyData** and **VerifyHash** methods use a public key to verify signatures computed using a private key. Both return a **bool** that indicates whether the verification is successful. Remember, the verification process only needs the public key from the RSA key pair.

Specifying Hash Algorithms

In .NET 4.6 things changed in terms of the method signatures for the **RSA** base class. Instead of specifying a hash algorithm name or an OID, the **HashAlgorithmName** is used to specify the hash algorithm. Table 26 describes the **HashAlgorithmName** members.

Table 26: HashAlgorithmName members

HashAlgorithmName (struct) member	Description
MD5	Gets a hash algorithm name for MD5.
SHA1	Gets a hash algorithm name for SHA1.

SHA256	Gets a hash algorithm name for SHA256.
SHA384	Gets a hash algorithm name for SHA384.
SHA512	Gets a hash algorithm name for SHA512.

RSA Signature Padding

The **RSASignaturePadding** class was added in .NET 4.6 to be used with RSA signature methods. This extends to the **RSA** base class as well as the two concrete classes: **RSACng** and **RSACryptoServiceProvider**. Table 27 contains the **RSASignaturePadding** members.

Table 27: RSASignaturePadding members

RSASignaturePadding member	Description
Pkcs1	Gets a Pkcs1 instance of **RSASignaturePadding**.
Pss	Gets a Pss instance of **RSASignaturePadding**.

Pkcs1 padding (PKCS#1 v1.5) has been used for years as a standardized method of padding. It uses a fixed value to pad data and is still considered secure.

Pss was added to the PKCS#1 standard in version 2.1. **Pss** generates a random salt value to use as part of the padding process, which provides increased security assurance over that of **Pkcs1**. While **Pkcs1** is still considered secure, you should start to transition to **Pss** where available and use **Pkcs1** for legacy.

Pre-.NET 4.6

In older versions of .NET the **RSA** base class does not have signature methods; they are only available through a concrete instance of the **RSACryptoServiceProvider** class. Keep in mind, the public key must be made available to the recipients for them to verify the private key signature.

In practice, either an **RSACryptoServiceProvider** or an **RSACng** instance will have to be used. For legacy applications, or for that matter anything before .NET 4.6, **RSACryptoServiceProvider** is the only option in .NET unless you bring in a third-party library.

Example: Sign and Verify Data with RSA

The **RSA** class is fairly intuitive. For construction and key information refer to the RSA section last chapter. This example shows how to sign and verify data using the **SignData** and **VerifyData** methods. The keys here are randomly generated in the **RSACng** instance, whereas typically you'll have a key pair. Notice that the **HashAlgorithmName** and **RSASignaturePadding** objects are used (.NET 4.6 and newer):

```
RSA rsa = new RSACng();

byte[] data = new byte[] { 0, 0, 0, 0 };
```

```
byte[] signedData = rsa.SignData(data, HashAlgorithmName.SHA256, RSASignaturePadding.Pss);

bool verify = rsa.VerifyData(data, signedData, HashAlgorithmName.SHA256, RSASignaturePaddin
g.Pss);
```

RSACryptoServiceProvider

The **RSACryptoServiceProvider** class is one of two concrete implementations of the **RSA** base class that exist as of .NET 4.6. Prior to .NET 4.6 it is the only concrete class that inherits from **RSA**.

.NET 4.6 and Newer

RSACryptoServiceProvider inherits from **RSA** and in turn the new changes that were made in .NET 4.6. These included the new encryption and signature methods and the new objects to be used with them, like **HashAlgorithmName**, **RSAEncryptionPadding** and **RSASignaturePadding**. The old style methods are still available in overloads and will have to be used in anything that is pre-.NET 4.6; these will be covered next section.

Pre-.NET 4.6

The older method signatures for signing and verification are a little more error-prone than the newer styles. Out of the two signing methods, **SignData** is more user-friendly. **SignHash** requires the developer to specify an Object Identifier (OID) that corresponds to a particular hash algorithm. These are not easy to remember. **SignData** will do the hashing for the developer and only requires a string name for the hash algorithm or an object reference, rather than the OID number. We will use **SignData** in our examples to avoid OID issues associated with the **SignHash** method. Additionally, **SignData** can process IO stream or byte-array data. The parameters call for the data and a hash algorithm object. The object representing the hash algorithm can either be a string representing the hash algorithm name, an instance of a **HashAlgorithm** object, or a **Type** representing a hash algorithm.

Below, a byte array of data is signed using the **SignData** method. The hash algorithm is specified using a string for the algorithm name (“SHA256”). Notice that the key pair is retrieved from the CryptoAPI store using a **CspParameters** object. Obviously, you can implement whatever style of key import or generation you’d like for the RSA keys.

```
CspParameters csp = new CspParameters();
csp.KeyContainerName = "rsaKeyPair";

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);

byte[] data = new byte[32];

byte[] signedData = rsa.SignData(data, "SHA256");
```

For a 1024-bit key (default in .NET) the signature will be 128 bytes in length, regardless of the hash algorithm.

The public key will need to be imported into an **RSA** object on the receiver’s end to verify the signature. Verification is performed using two methods analogous to those we just covered: **VerifyData** and **VerifyHash**. These are intended to be used with the **SignData** and **SignHash** methods, respectively, and also take the same parameters in addition to the digital signature that must be verified. Both return a bool value affirming or denying verification. To coincide with our last example, we will use the **VerifyData** method to verify the data.

```
bool match = rsa.VerifyData(data, "SHA256", signedData);
```

Example: Simple RSACryptoServiceProvider Verification (Pre-.NET 4.6)

Here is an example of a simple method to verify a signature by importing a **CspBlob** containing a party's public key. The plaintext message must be provided in the *data* parameter. The received signature must be supplied in the *signature* parameter. A string parameter is included for the hash algorithm name. Notice that the public key must be imported, which will always be the case if you are verifying a third-party signature.

```
public bool VerifyData(byte[] data, byte[] signature, byte[] publicKey, string
hashAlgorithm)
{
    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider())
    {
        rsa.ImportCspBlob(publicKey);

        return rsa.VerifyData(data, hashAlgorithm, signature);
    }
}
```

Last chapter covers importing and exporting keys with **CspParameters** and **RSACryptoServiceProvider**.

Example: Using CryptoConfig to map a hash OID

In chapter 2, we covered the basic usages of the **CryptoConfig** class and how to perform basic mapping between objects, names, and object identifiers (OIDs). The **SignHash** method in pre-.NET 4.6 **RSACryptoServiceProvider** requires an OID to specify the hash algorithm. We have a couple of options here: lookup the OID and manually enter it by hand, or use the **CryptoConfig** class to map our algorithm name to the OID. In this example we'll use the latter: **CryptoConfig.MapNameToOID**. The example code below shows how this can be done:

```
byte[] data = new byte[32];

byte[] hash;

using (var sha = new SHA256Cng())
    hash = sha.ComputeHash(data);

byte[] signature;

string oid = CryptoConfig.MapNameToOID("SHA256");

using (var rsa = new RSACryptoServiceProvider())
    signature = rsa.SignHash(hash, oid);
```

RSACng

The **RSACng** class was added in .NET 4.6 and inherits from **RSA**. **RSACng** does not offer the older style encryption and signing methods that **RSACryptoServiceProvider** does. This being the case, **RSACng** will not be fully compatible with older .NET solutions.

Signing and verification methods are inherited from the new **RSA** base class and will work like described in the **RSA** section. Remember, the **HashAlgorithmName** and **RSASignaturePadding** objects must be congruent between the signing and verification processes (these are listed in the first section of the chapter). For example, if the signing is performed with SHA1, specifying SHA256 during verification will throw an error.

DSA

DSA (Digital Signature Algorithm) is a FIPS approved public key algorithm for computing and verifying digital signatures. The DSA standard uses the SHA1 hash algorithm. This is a concern for secure systems that need to maintain a 128-bit security strength, as SHA1 offers less than 80 bits of collision resistance. Like RSA, DSA must use a public/private key pair (however, RSA and DSA cannot share keys). Key pairs can be randomly generated on the fly, or retrieved from persistent storage like a key store or a secured file.

The DSA Base Class

The **DSA** abstract base class contains two signature related methods: **CreateSignature** and **VerifySignature**. The static **Create** method will create an instance of **DSACryptoServiceProvider**, the only concrete **DSA** implementation:

```
DSA dsa = DSA.Create();
```

Example: Sign and Verify with DSA

Only SHA1 hashes can be signed with DSA; there are no methods for signing data itself, like with RSA. The **CreateSignature** and **VerifySignature** methods are used below with a default instance of **DSA**. The hash is computed using **SHA1Managed**:

```
byte[] data = new byte[32];
byte[] hash = null;

using (var sha1 = new SHA1Managed())
    hash = sha1.ComputeHash(data);

DSA dsa = DSA.Create();

byte[] signature = dsa.CreateSignature(hash);

bool verified = dsa.VerifySignature(hash, signature);
```

DSACryptoServiceProvider

At the time of this writing, **DSACryptoServiceProvider** is the only concrete implementation of the **DSA** abstract class. **DSACryptoServiceProvider** can use the same general formats as the **RSACryptoServiceProvider** for storage and transmitting keys like CspBlob, XML string, and **CspParameters**. **DSAParameters** objects (analogous to the **RSAParameters** object with **RSACryptoServiceProvider**) can be used to access the key information associated with the **DSA** algorithm instance. We won't include examples for using DSA keys in the formats that they share with RSA. However, we will cover how to use **CspParameters** to persist keys to the CryptoAPI key store (**CspParameters** and **CspParameters** flags were covered last chapter in the **RSACryptoServiceProvider** section).

---

*DSA Bug in .NET 1.0 and 1.1: You cannot inherit from **DSA** outside of mscorlib.dll in .NET framework versions 1.0 and 1.1 because the constructor is defined as internal.*

Creating and Verifying Digital Signatures with DSACryptoServiceProvider

DSACryptoServiceProvider can sign raw or hashed data depending on which of its methods are used: **SignData** signs raw data; **SignHash** signs a cryptographic hash value. The methods for verification work the same way. We will only provide a quick example here because the relevant info for signing and verification from the **RSACryptoServiceProvider** signature examples will carry over.

Signing with the **SignData** method:

```
DSACryptoServiceProvider dsa = new DSACryptoServiceProvider();

byte[] data = Encoding.UTF8.GetBytes("This is some data");

byte[] signature = dsa.SignData(data);
```

Verification will require the data and the digital signature in question. Verifying with **VerifyData**:

```
byte[] verified = dsa.VerifyData(data,signature);
```

***Hash Algorithms with DSA**: SHA1 is the standard algorithm for DSA and the only accepted algorithm in the .NET **DSA** implementations and the **DSASignature[De]Formatter** classes. By default, the **SignatureAlgorithm** property of **DSACryptoServiceProvider** will return "http://www.w3.org/2000/09/xmldsig#dsa-sha1" as its signature algorithm.*

Using CspParameters with DSACryptoServiceProvider

At the time of this writing, .NET uses an RSA provider type -24- in **CspParameters** objects by default. DSA implementations will throw errors if their **CspParameters** object is not set to the correct provider type. Provider types are easily set through the **CspParameters.ProviderType** property. The default provider type that .NET uses with **DSACryptoServiceProvider** is “13”, “Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider”.

Setting a provider type is straightforward and only adds one additional line of code to the **CspParameters** examples from page 128. Below, the key container name is referenced as usual, but the provider type is explicitly set to 13:

```
CspParameters csp = new CspParameters();
csp.KeyContainerName = "dsaKeyPair";
csp.ProviderType = 13;

DSACryptoServiceProvider dsa = new DSACryptoServiceProvider(csp);
```

Last chapter contains additional information on using **CspParameters** objects.

DSASignature[De]Formatter

These classes are similar to the [de]formatter classes used with the RSA key exchange. Developers are usually encouraged to use these classes to achieve better portability over the concrete DSA implementations, helping lower dependencies. They also are designed to resist attacks that might compromise the raw implementation of DSA that are incorrect. The **DSASignature[De]Formatter** classes consume a DSA algorithm object that implements **AsymmetricAlgorithm**. At the time of this writing, **DSACryptoServiceProvider** is the only such class in .NET that meets these criteria. In other words, **DSASignature[De]Formatter** is currently only usable with **DSACryptoServiceProvider**.

The **[De]Formatter** classes use the DSA PCKS #1 signature format. While portability is enhanced through these objects, their interfaces are limited (yet simpler) compared to the **DSACryptoServiceProvider**. Here, you must hash the data first before signing.

An **AsymmetricAlgorithm** must be set through the constructor or the **SetKey** method before signing or verification can take place. Below we use a **DSACryptoServiceProvider** object, which gets its keys from a **CspParameters** object, to provide keys to **DSASignatureFormatter** through its constructor:

```
CspParameters csp = new CspParameters();
csp.KeyContainerName = "dsaKeyPair";
csp.ProviderType = 13;

DSACryptoServiceProvider dsa = new DSACryptoServiceProvider(csp);

DSASignatureFormatter formatter = new DSASignatureFormatter(dsa);
```

Alternatively, the **SetKey** method will also work:

```
formatter.SetKey(dsa);
```

Creating a signature with the **DSASignatureFormatter** class can only take place after the data is hashed. Below, data is hashed and the hash is signed using the formatter's **CreateSignature** method (the formatter object is assumed to get its DSA reference from the last example):

```
byte[] data = Encoding.UTF8.GetBytes("This is some data");

byte[] hash = new SHA1Managed().ComputeHash(data);

DSASignatureFormatter formatter = new DSASignatureFormatter(dsa);

byte[] sig = formatter.CreateSignature(hash);
```

Full verification of the data requires the original data in the format in which it was hashed, the hash of the data (SHA1), and the hash's signature—plus the sender's public key. The verification process is carried out in a fashion essentially opposite that of the creation process:

```
byte[] sig=...

byte[] data = Encoding.UTF8.GetBytes("This is some data");

byte[] hash = new SHA1Managed().ComputeHash(data);

//Assumes DSA object contains the sender's public key
DSASignatureDeformatter deformatter = new DSASignatureDeformatter(dsa);

bool verify = deformatter.VerifySignature(hash, sig);
```

Only the data used to create the hash needs to be transmitted/persisted, the hash does not as long as both parties hash the data in the same manner.

Example: Signing and Verification using DSASignature[De]Formatter

This example will use the **DSASignature[De]Formatter** classes to format a digital signature. We use the **DSACryptoServiceProvider** class to provide the algorithm instance to the formatter objects.

The first method, *Sign*, uses an instance of **AsymmetricAlgorithm** (the base class for asymmetric algorithm objects, such as RSA or DSA) to create the **DSASignatureFormatter** object, which in turn formats the signature for the SHA1 hash:

```
public byte[] Sign(byte[] data, AsymmetricAlgorithm alg)
{
    byte[] hash = new SHA1Managed().ComputeHash(data);

    DSASignatureFormatter formatter = new DSASignatureFormatter(alg);

    return formatter.CreateSignature(hash);
}
```

Verify, the next method, hashes the incoming data and verifies the signature using a **DSASignatureDeFormatter** object instantiated with an **AsymmetricAlgorithm** instance:

```
public bool Verify(byte[] data, byte[] signature, AsymmetricAlgorithm alg)
{
    DSASignatureDeformatter deformatter = new DSASignatureDeformatter(alg);

    byte[] hash = new SHA1Managed().ComputeHash(data);

    return deformatter.VerifySignature(hash, signature);
}
```

Now we'll give it a try using an instance of **DSACryptoServiceProvider** to give us a key pair, which could represent, in a simple context, the signer (the private key) and the recipient (the public key):

```
DSACryptoServiceProvider dsa = new DSACryptoServiceProvider();

byte[] message = new byte[32];

byte[] signature = Sign(message, dsa);

bool ok = Verify(message, signature, dsa);
```

DSASignature[De]Formatter Compatibility*: At the time of this writing (.NET 4.6), DSACryptoServiceProvider is the only DSA subclass that will work with the DSASignature[De]Formatter objects.*

Elliptic Curve DSA and the ECDsa Base Class

Elliptic Curve (EC) algorithms are more efficient and have comparatively smaller key sizes than regular public key algorithms. As a result, they are often better suited for mobile applications. This, however, should not bar them from being used in other types of applications as well. EC algorithms should be preferred in most cases besides legacy compatibility.

.NET defines an **ECDsa** abstract class for the Elliptic Curve DSA algorithm. Its functionality is less than that of its concrete class, **ECDsaCng**. **ECDsa** only defines two methods for digital signing: **SignHash** and **VerifyHash**. **ECDsa** behaves much like many of the other classes that perform digital signing, and could even be considered simpler. For one, there are no hash names, OIDs, or padding styles to specify.

Example: Simple Sign and Verify with ECDsa

Here we will use the **SignHash** and **VerifyHash** methods of **ECDsa** to sign and verify a hash using a randomly generated algorithm instance. These methods are very straightforward and only require a hash value for signing, and a hash and a signature for verification.

```
var data = new byte[40];

var hash = new SHA256Managed().ComputeHash(data);

bool verified = false;

using (ECDsa ecDsa = new ECDsaCng())
{

    var signature = ecDsa.SignHash(hash);

    verified = ecDsa.VerifyHash(hash, signature);
}
```

ECDsaCng

ECDsaCng provides a Cryptography Next Generation (CNG) implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA).

By now you've observed the similarities between the interfaces of different digital signature algorithms. **ECDsaCng** is similar to the others you've used and implements the **ECDsa** and **AsymmetricAlgorithm** abstract classes. Like the **RSACryptoServiceProvider** and **DSACryptoServiceProvider** implementations, **ECDsaCng** can sign and verify data through its **SignHash**, **SignData**, **VerifyHash**, and **VerifyData**, methods.

CNG Keys: CngKey objects provide a means to create, store, format, and open keys to use with different (Cryptography Next Generation) CNG algorithms.

Unless we want to use an ephemeral key (a short term, or session key) we need to access a key pair that we have stored. This could be a key imported from a file in an XML or blob type format, or a **CngKey** object that gets its keys from the key store. Below, we go with the key store option and create an instance of **ECDsaCng** with the **CngKey** in the constructor:

```
CngKey ecDsaKey = null;

if (CngKey.Exists("ecDsa521Pair"))
{
    ecDsaKey = CngKey.Open("ecDsa521Pair");
}
else
{
    ecDsaKey = CngKey.Create(CngAlgorithm.ECDsaP521,"ecDsa521Pair");
}

ECDsaCng ecDsa = new ECDsaCng(ecDsaKey);
```

(If you are lost as to what is going on above with opening and creating keys, read the *Working with Cryptography Next Generation (CNG) Keys* section in the last chapter) Next, we'll use the instance we created above to create a digital signature using the **SignData** method and verify the same signature with the **VerifyData** method:

```
byte[] data = Encoding.UTF8.GetBytes("Data!");

byte[] signature = ecDsa.SignData(data);

bool verified = ecDsa.VerifyData(data, signature);
```

ECDsa Signature Lengths

Table 28 contains the expected signature lengths that correspond to the type of ECDSA CNG key. P521 is the strongest key length and is currently the default in .NET 4.6.

Table 28: ECDSA Signature lengths

Algorithm/Key	Signature Length (bytes)
P256	64
P384	96
P521	132

XML Signatures

The XML Signature specification, also known as XMLDSIG, is a standardized method for digitally signing XML documents. .NET delivers this functionality in the **System.Security.Cryptography.Xml** namespace within the **System.Security.dll**. We will not include examples of XML signatures in this book for two reasons. First, some new cryptographic objects in .NET 4.6 are not compatible with what is considered to be standard usage of XML signatures. Second, the specifics of programming XML signatures in .NET revolve more around the particulars of the .NET XML classes than any type of specialized cryptographic objects.

On a side note, I personally think that XMLDSIG is a problematic answer to digital signing; I typically favor a simpler signature solution, even if I'm handling XML (which I try not to do).

Self-Verification of Signed Data

In an asymmetric model where we are encrypting data, we are not assumed to possess the private key. But when digitally signing, the signer is assumed to possess both the public and private keys. Because the signer has both keys, he or she can verify the signature is correct before sending it. This helps catch mathematical or programming errors in the signing process. Mathematical errors are far less common than programming errors, but they do happen. Checking your signature can also protect against attacks that seek to tamper with a private key or tamper with the signing process itself. Obviously this will not be much help if both of the keys or the entire signing process are compromised, but it's a good practice nonetheless, especially so for high-security or mission-critical systems.

Below, we have an example of an **RSACryptoServiceProvider** signing function that checks its signature before it's sent. The **CspParameters** object gives **RSACryptoServiceProvider** access to both of Alice's keys and makes verification a very easy and logical step of the signing process. A **CryptographicException** is thrown in the event the verification fails:

```
public byte[] AliceSignData(byte[] data)
{
    CspParameters csp = new CspParameters();
    csp.KeyContainerName = "aliceRsaKeyPair";

    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp))
    {
        byte[] signedData= rsa.SignData(data, "SHA256");

        if (rsa.VerifyData(data, "SHA256", signedData))
        {
            return signedData;
        }
        else
        {
            throw new CryptographicException("Signature Error.");
        }
    }
}
```

Applying the Horton Principle to Digital Signing

We first covered the Horton Principle on page 116. The Horton Principle tells us that we should authenticate *what is meant* in a communication, not just *what is said*. This is not just limited to symmetric key MACs. It should be a concern in all aspects of authentication involving messages and communication channels. If a condition is assumed to be present to adequately enforce security, it should be authenticated. For example, if we are engaging in secure communications with another person, and we know that they should be using a particular station, in a particular room, at a particular building, via a particular IP, we should authenticate these details in addition to just the message content.

A more detailed list of factors to be authenticated can be found on page 116. This is not an absolute list, just a starting point. You should authenticate whatever aspects of your channel are necessary to enforce your security model.

Asymmetric Encryption and Digital Signing: *Signcryption*

The combination of public-key encryption and signing is often referred to as *signcryption*. Many existing protocols use an encrypt-and-authenticate approach to digital signing where a signature is produced from the sensitive data and subsequently attached to the ciphertext after the data has been encrypted.

Unless you are interfacing with an existing protocol that dictates at what level and in which order the data is to be signed, you can order signing and encryption as you see fit. As expected, this must be congruent between the sender and receiver.

Hybrid encryption—encrypting the data with a symmetric algorithm and securing the symmetric key with asymmetric encryption—is still recommended even where public-key encryption and signing are combined.

Example: RSACng Hybrid Signcryption

This example will combine digital signing with the hybrid encryption that we covered last chapter. Encrypt-and-Authenticate will be used as the order for encryption and signing. Here, the plaintext message will be signed with the sender's private-key, and the encryption will be performed with the receiver's public-key. The signature is attached (in this case appended) to the ciphertext. (Next section will cover the different orders of encryption and signing).

We will write two methods: one for encryption and signing; and another for decryption and verification. The AES *Encrypt* and *Decrypt* methods from page 89 will be used to perform symmetric encryption and decryption (here they are called *SymmetricEncrypt* and *SymmetricDecrypt*).

The first method, *EncryptAndSign*, specifies sender and receiver instances of **RSACng**, and the data (message) as parameters. The actual code progresses as follows:

- A new 32-byte symmetric key is generated
- The data is encrypted with the *SymmetricEncrypt* method
- The signature is created using the **SignData** method of **RSACng**. **SHA256** and **Pss** are used as the **HashAlgorithmName** and **RSASignaturePadding**, respectively.
- The **Encrypt** method of the receiver's **RSACng** instance is used to encrypt the randomly generated symmetric key.
- The encrypted (secured) symmetric key, symmetric ciphertext, and the signature are concatenated in this order and returned (in this scenario to be transmitted to the other party).

```
byte[] EncryptAndSign(RSACng signKey, RSACng encryptKey, byte[] data)
{
    byte[] symmetricKey = new byte[32];

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(symmetricKey);

    byte[] ciphertext = SymmetricEncrypt(data, symmetricKey);

    byte[] signature =
        signKey.SignData(data, HashAlgorithmName.SHA256, RSASignaturePadding.Pss);

    byte[] securedKey = encryptKey.Encrypt(symmetricKey, RSAEncryptionPadding.OaepSHA256);

    return securedKey.Concat(ciphertext).Concat(signature).ToArray();
}
```

Next is the second method: *DecryptAndVerify*. Both parties' **RSACng** objects must still be supplied as arguments as well as the ciphertext (presumably from *EncryptAndSign*). The process for decryption and verification essentially operates in opposite order of the last method:

- First, we will convert the key sizes to bytes in order to obtain the correct data offsets/positions for parsing.
- Next we will actually extract the sections of the byte array that contain the secured key, the symmetric ciphertext, and the signature.

- The secured key is decrypted using the receiver's **RSACng** instance to obtain the symmetric key.
- The symmetric key is used to decrypt the ciphertext.
- Lastly, the sender's algorithm instance is used to verify the received plaintext after decryption. The method returns the plaintext if verification is successful and a null otherwise.

```
byte[] HybridDecryptAndVerify(RSACng decryptKey, RSACng verifyKey, byte[] data)
{
    //Convert key sizes in bits to bytes using shift.
    int verifyKeySize = verifyKey.KeySize >> 3;
    int decryptKeySize = decryptKey.KeySize >> 3;

    byte[] securedKey = data.Take(decryptKeySize).ToArray();

    byte[] signature = data.Skip(decryptKeySize).Take(verifyKeySize).ToArray();

    byte[] ciphertext = data.Skip(verifyKeySize + decryptKeySize).ToArray();

    byte[] symmetricKey = decryptKey.Decrypt(securedKey, RSAEncryptionPadding.OaepSHA256);

    byte[] plaintext = Decrypt(ciphertext, symmetricKey);

    if (verifyKey.VerifyData(plaintext, signature, HashAlgorithmName.SHA256, RSASignaturePa
dding.Pss))
        return plaintext;
    else
        return null;
}
```

Let's give these methods a trial run. Two new algorithm instances are created below to represent Bob and Alice:

```
//Create "instances" of Bob and Alice.
var alice = new RSACng(4096);
var bob = new RSACng(1024);

//The message.
byte[] data = new byte[400];

//Bob sends data to Alice.
byte[] securedMessage = EncryptAndSign(bob, alice, data);

//Alice receives from Bob.
byte[] verifiedMessage = DecryptAndVerify(bob, alice, securedMessage);
```

Order and Scope of Digital Signing and Encryption

We first covered the three different orders in which to perform message authentication on page 109:

- Encrypt-and-Authenticate (Encrypt-and-Sign)
- Authenticate-then-Encrypt (Sign-then-Encrypt)
- Encrypt-then-Authenticate (Encrypt-then-Sign)

We can do the same thing with digital signatures and public-key encryption.

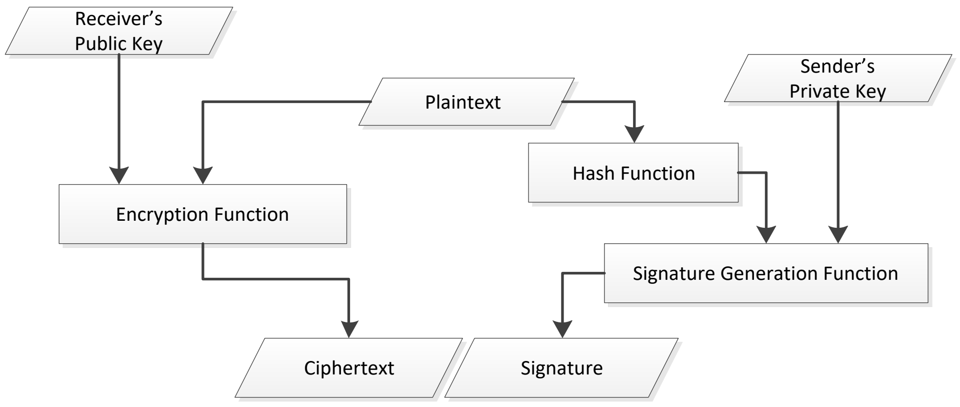
The asymmetric model of encryption and authentication does not assume the keys to be pre-shared, as with the symmetric key model. In a public key environment we have different keys with different purposes depending on the context. The following examples show different orders of encryption and authentication in a public-key environment as well as which keys are used in which context.

Encrypt-and-Authenticate (Encrypt-and-Sign)

The encrypt-and-authenticate approach to signcryption encrypts and digitally signs the plaintext message. In this approach both encryption and digital signing can be parallelized on the sender's end. The receiver will have to decrypt first before verifying the signature.

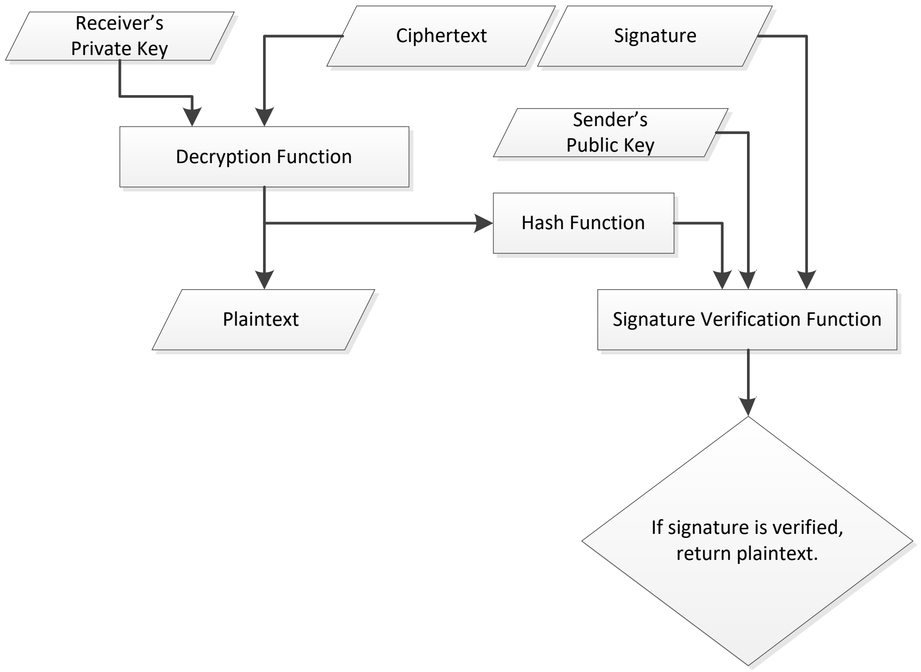
This is the only signcryption model out of the three in this section that provides a truly direct means of nonrepudiation, which is why it is very popular. A signature can be verified against a message without involving any receiving party's keys. Figure 38 and Figure 39 illustrate the encrypt-and-authenticate process as the sender and receiver, respectively.

Figure 38: Encrypt-and-Authenticate (Encrypt-and-Sign) (sender)



```
byte[] EncryptAndSign(RSACng signKey, RSACng encryptKey, byte[] data)
{
    byte[] symmetricKey = new byte[32];

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(symmetricKey);

    byte[] ciphertext = SymmetricEncrypt(data, symmetricKey);

    byte[] signature =
        signKey.SignData(data, HashAlgorithmName.SHA256,
RSASignaturePadding.Pss);

    byte[] securedKey = encryptKey.Encrypt(symmetricKey,
RSAEncryptionPadding.OaepSHA256);

    return securedKey.Concat(ciphertext).Concat(signature).ToArray();
}
```

Figure 39: Encrypt-and-Authenticate (Encrypt-and-Sign) (receiver)



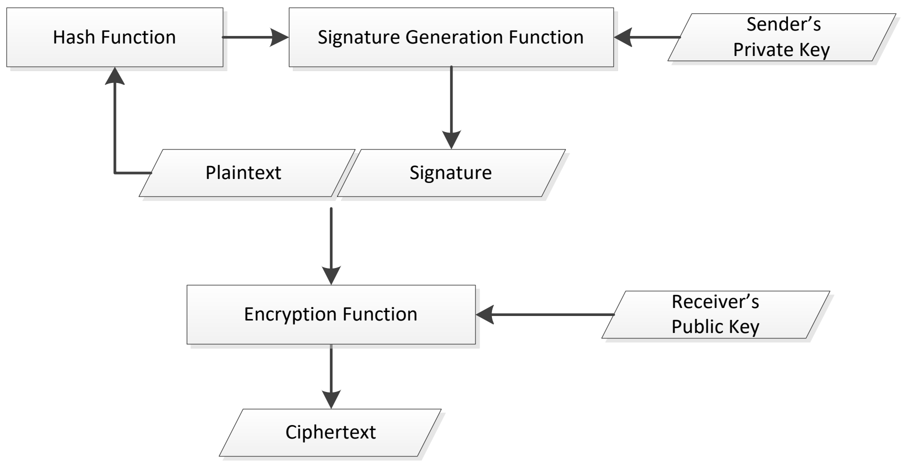
```
byte[] DecryptAndVerify(RSACng verifyKey, RSACng decryptKey, byte[] data)
{
    //Convert key sizes in bits to bytes using shift.
    int verifyKeySize = verifyKey.KeySize >> 3;
    int decryptKeySize = decryptKey.KeySize >> 3;

    byte[] securedKey = data.Take(decryptKeySize).ToArray();

    byte[] signature =
data.Skip(data.Length - verifyKeySize).ToArray();

    byte[] ciphertext =
data.Skip(decryptKeySize).Take(data.Length - (decryptKeySize +   verifyKeySize)).ToArray();

    byte[] symmetricKey =
     decryptKey.Decrypt(securedKey, RSAEncryptionPadding.OaepSHA256);

    byte[] plaintext = SymmetricDecrypt(ciphertext, symmetricKey);

    if (verifyKey.VerifyData(plaintext, signature, HashAlgorithmName.SHA256, RSASignaturePa
dding.Pss))
        return plaintext;
    else
        return null;
}
```

Authenticate-then-Encrypt (Sign-then-Encrypt)

Authenticate-then-Encrypt generates a signature of the plaintext message and encrypts the message as well as the signature. Unlike encrypt-and-sign, which can be parallelized on the sender's end, this approach can be parallelized on the receiver's end (decrypt/verify).

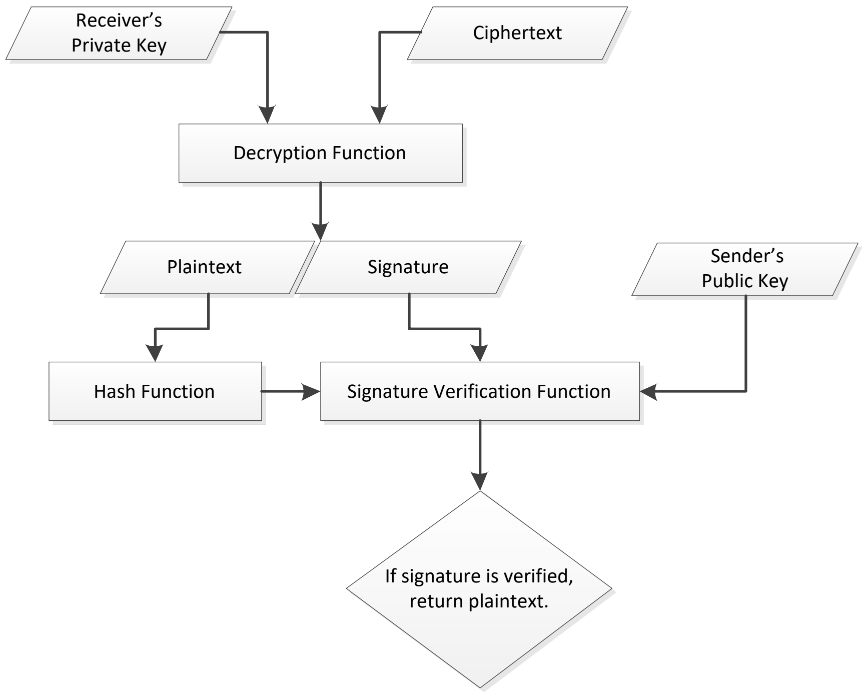
Direct nonrepudiation cannot be achieved in this model because the receiving party's keys must be implicated. Encrypt-and-Authenticate (encrypt-and-sign) should be used where true nonrepudiation is required. Figure 40 and Figure 41 illustrate the authenticate-then-encrypt process as the sender and receiver, respectively.

Figure 40: Authenticate-then-Encrypt (Sign-then-Encrypt) (sender)



```
byte[] SignThenEncrypt(RSACng signKey, RSACng encryptKey, byte[] data)
{
    byte[] symmetricKey = new byte[32];

    byte[] signature =
        signKey.SignData(data, HashAlgorithmName.SHA256, RSASignaturePadding.Pss);

    byte[] signedData = data.Concat(signature).ToArray();

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(symmetricKey);

    byte[] ciphertext = SymmetricEncrypt(signedData, symmetricKey);

    byte[] securedKey = encryptKey.Encrypt(symmetricKey, RSAEncryptionPadding.OaepSHA256);

    return securedKey.Concat(ciphertext).ToArray();
}
```

Figure 41: Authenticate-then-Encrypt (Sign-then-Encrypt) (receiver)



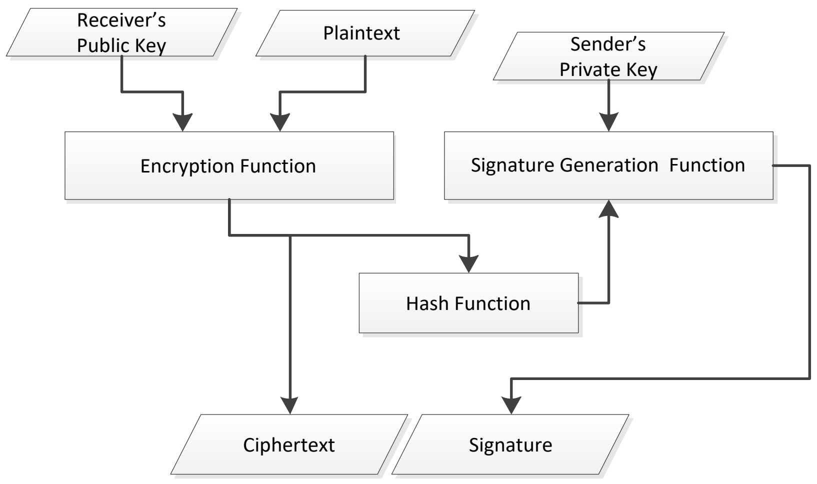
```
byte[] DecryptThenVerify(RSACng verifyKey, RSACng decryptKey, byte[] data)
{
    //Convert key sizes in bits to bytes using shift.
    int verifyKeySize = verifyKey.KeySize >> 3;
    int decryptKeySize = decryptKey.KeySize >> 3;

    byte[] securedKey = data.Take(decryptKeySize).ToArray();

    byte[] ciphertext = data.Skip(decryptKeySize).ToArray();

    byte[] symmetricKey = decryptKey.Decrypt(securedKey, RSAEncryptionPadding.OaepSHA256);

    byte[] signedData = SymmetricDecrypt(ciphertext, symmetricKey);

    byte[] plaintext = signedData.Take(signedData.Length - verifyKeySize).ToArray();

    byte[] signature = signedData.Skip(signedData.Length - verifyKeySize).ToArray();

    if (verifyKey.VerifyData(plaintext, signature, HashAlgorithmName.SHA256, RSASignaturePa
dding.Pss))
        return plaintext;
    else
        return null;
}
```

Encrypt-then-Authenticate (Encrypt-then-Sign)

Encrypt-then-authenticate encrypts the plaintext message and generates a signature of the ciphertext. Similar to authenticate-then-encrypt, neither the sender's or receiver's side of the process can be parallelized.

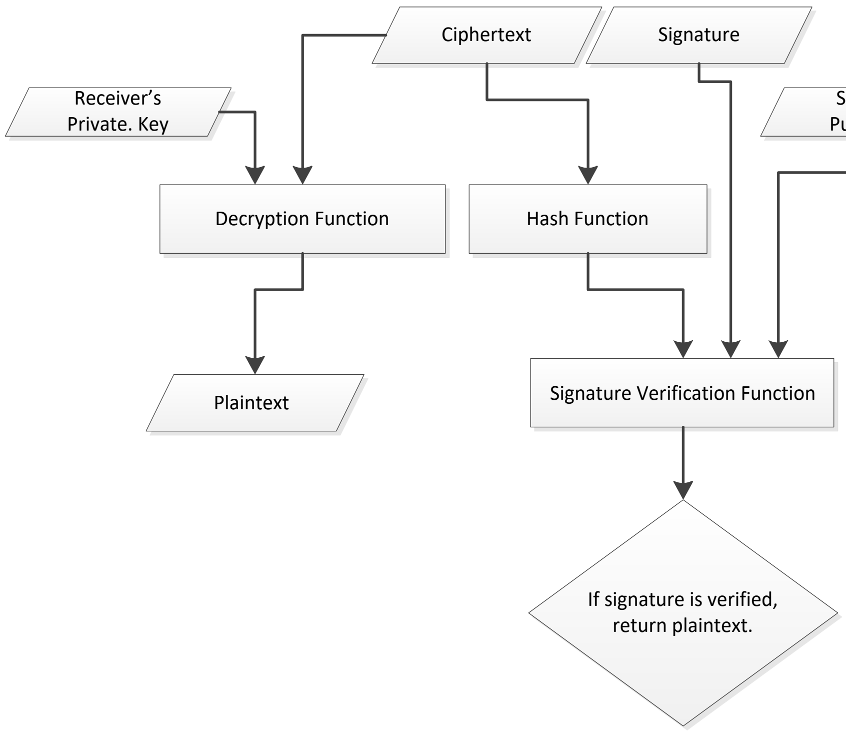
A problem native to this model is that it cannot provide direct nonrepudiation based on just the message and the signer's public key. This is because the ciphertext is what is actually being signed, not the message itself, which means that the receiving party is always going to be implicated in the verification process. Encrypt-and-Authenticate (encrypt-and-sign) should be used where true nonrepudiation is required. Figure 42 and Figure 43 illustrate the encrypt-then-authenticate process as the sender and receiver, respectively.

Figure 42: Encrypt-then-Authenticate (Encrypt-then-Sign) (sender)



```
byte[] EncryptThenSign(RSACng signKey, RSACng encryptKey, byte[] data)
{
    byte[] symmetricKey = new byte[32];

    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(symmetricKey);

    byte[] ciphertext = Encrypt(data, symmetricKey);

    byte[] securedKey = encryptKey.Encrypt(symmetricKey, RSAEncryptionPadding.OaepSHA256);

    byte[] securedKeyAndCiphertext = securedKey.Concat(ciphertext).ToArray();

    byte[] sig = signKey.SignData(securedKeyAndCiphertext, HashAlgorithmName.SHA256, RSASig
naturePadding.Pss);

    return securedKeyAndCiphertext.Concat(sig).ToArray();
}
```

Figure 43: Encrypt-then-Authenticate (Encrypt-then-Sign) (receiver)



```
byte[] VerifyThenDecrypt(RSACng verifyKey, RSACng decryptKey, byte[] data)
{
    //Convert key sizes in bits to bytes using shift.
    var decryptKeySize = decryptKey.KeySize >> 3;
    var signKeySize = verifyKey.KeySize >> 3;

    byte[] securedKeyAndCiphertext = data.Take(data.Length - signKeySize).ToArray();

    byte[] sig = data.Skip(data.Length - signKeySize).Take(signKeySize).ToArray();

    bool verified = verifyKey.VerifyData(securedKeyAndCiphertext, sig, HashAlgorithmName.SH
A256, RSASignaturePadding.Pss);

    if (verified)
    {
        byte[] symmetrickey = decryptKey.Decrypt(securedKeyAndCiphertext.Take(decryptKeySiz
e).ToArray(), RSAEncryptionPadding.OaepSHA256);

        byte[] ciphertext = securedKeyAndCiphertext.Skip(decryptKeySize).Take(data.Length -
 (decryptKeySize + signKeySize)).ToArray();

        return SymmetricDecrypt(ciphertext, symmetrickey);
    }
    else
    {
        return null;
    }
}
```

Recommendations

In terms of security, we recommend the encrypt-*and*-authenticate (encrypt-*and*-sign) scheme because it offers true nonrepudiation (remember, this is different from the encrypt-*then*-sign). Otherwise, you may have to implement a scheme dictated by the protocol you are implementing. Encrypt-and-authenticate is also parallelizable, offering better performance during the encryption and signing stage.

Hardening Signcryption Schemes

Security is the most important factor when deciding which approach to use and how you will write it specifically. Our security goals are simple:

1. Privacy: Only the legitimate sender *S* and receiver *R* should know the contents of a message *M.*
2. Integrity: The receiver *R* should have assurance that *M* was sent from *S*.

But won't encryption (privacy) and digital signatures (integrity) provide the necessary assurance? Not always. Clever attack models can be used to compromise these protections and specific steps can be taken to harden an encryption/signature scheme against them.

Chosen-Plaintext Attack (CPA) on Authenticate-then-Encrypt

A CPA attack can be used to attack the authenticate-then-encrypt model. In this attack the attacker *A* poses as malicious receiver and receives a message from a sender *S*. *A* is able to decrypt the contents and view the message and signature. Now, *A* is able to encrypt the contents and send them to a legitimate receiver *R* using *R's* public key. Depending on how *A* wants to take advantage of the situation, it could easily sit contently as the man-in-the-middle between a legitimate sender and receiver, or view the contents and send the message to another receiver *R* where a contextual change could have huge implications.

Chosen-Ciphertext Attack (CCA) on Encrypt-then-Authenticate

A CCA can exploit the encrypt-then-authenticate model where an attacker *A* can observe and intercept a message *M* from a legitimate sender S to a receiver *R*. Because the encrypt-then-authenticate model signs the ciphertext rather than the message, the attacker is able to replace the original signature with their own and pose as the sender. *R* now considers *A* to be the legitimate sender. Keep in mind that *A* still doesn't know the contents of *M*. To get the contents to *M*, *A* could try a few things:

1. Request a summary of the conversation (*R* sees this as the legitimate sender/party requesting the conversation).
2. Request that *R* forward *M* to another party/account that *A* has full control over.
3. Try to maintain the conversation and indirectly glean as much data as possible about *M*, assuming that the context or details can divulge this.

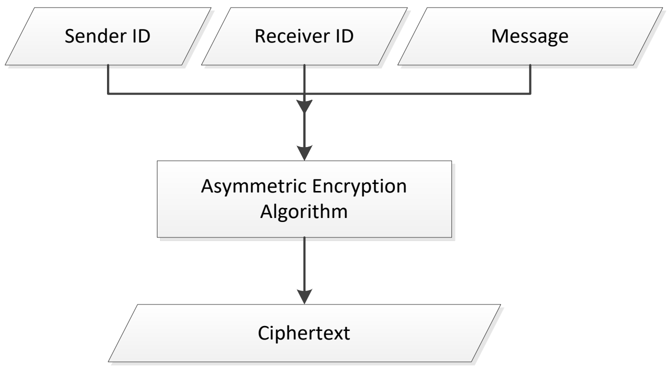
Past this, the attack also poses a serious threat in terms of the attacker *A* being able to manipulate the conversation, seeking to change the details of the message *M* originally sent by the legitimate sender *S*. For example, if *A* knows that *M* pertains to a request to transfer funds to an account, it could add "Actually, transfer the funds to account ### instead and make the amount $1,000,000 USD. Thank you."

Using Identifiers

To fix the CCA and CPA issues discussed above, identifiers should be implemented which identify the sender and receiver, such as a name or ID number. These should be encrypted with the message, and included with the signature generation process. Upon decryption the receiver will be able to verify the included identities for the sender and themselves, and verify that the signature verifies with these values. This ensures that the signature as well as the ciphertext include relevant identities.

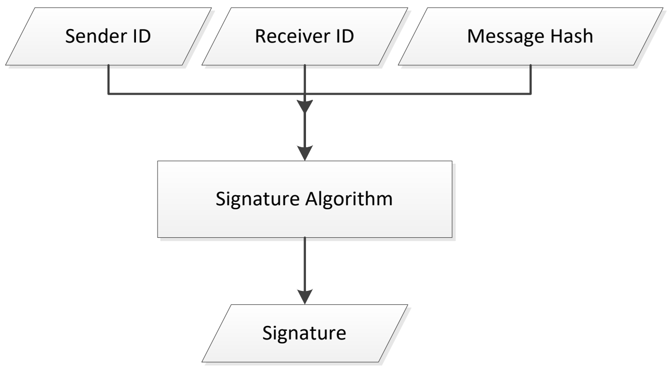
In Figure 44, identifiers for the sender and receiver IDs as well as the message are passed into an asymmetric algorithm.

Figure 44: Using Identifiers in Asymmetric Encryption



In Figure 45, the identifiers and the message (or the ciphertext, depending on what you are signing) are passed into a signature algorithm.

Figure 45: Using Identifiers in Digital Signing



Replay Attacks

Replay attacks retransmit a previously transmitted message. In other words, the attacker is simply reusing an old valid message that was intercepted. This may seem trivial but it can be devastating. Imagine where an attacker is able to capture a message where a legitimate party is transmitting the attacker some money. If it can freely replay this message, and it is accepted by the receiver as legitimate, the attacker can get lots of free money from some unlucky owner of an account.

Like the vulnerabilities in the last section, replay attacks too can be fixed using a type of identifier called a sequence number, or message number. Instead of identifying the parties (which we should still do), this identifies the message itself. Most of the time this is a sequential number that indicates if a message is in the correct order. And like the other identifiers, sequence numbers should also be included in encryption and signing.

Implementation Issues

Strong Trust

First, let's look at *Strong Trust* with a Certificate Authority. *Strong Trust* is established through a trusted CA. This allows us to trust that we are talking to the person we think we are talking to. On a large scale this would be an entity like Verisign. We trust *Verisign* to vouch for the identity of entities that we communicate with. On a smaller scale this could be a company CA that establishes trust that an identity within the organization is who they say they are. They cannot repudiate because we have their signature and we trust the CA who has confirmed their identity. Most importantly we know that we are communicating with the right person.

Other types of trust are also possible without the use of a CA. By definition, however, these do not provide "Strong Trust". One option is having a hardcoded list of trusted entities, such as a client keeping a list of trusted server public keys. This type of trust is built in by the developer without external interaction with a CA. If the server key is not on the list it isn't trusted. However, this type of solution can have availability and scalability issues. What if a server key is compromised? The hardcoded list in the application is no longer trustworthy until the app is rebuilt. Until then, clients are in serious danger of turning over sensitive information to an attacker who controls the server or possesses the private key.

What this tells you is that without trust, technology cannot protect us. There must be trust. Public Key Infrastructure (PKI) was created to verify the identity of—establish trust in—otherwise remote, untrusted parties. Just because you don't use a CA, doesn't mean that you can't establish trust, it depends on the context and the level of trust that is expected or needed.

Formatting

When designing applications that use digital signatures, formatting will need to be performed consistently to attach and remove the digital signatures from the data. This needs to be a robust mechanism to appropriately handle malformed or broken formatting.

Compatibility

Compatibility is a very real concern when implementing production security systems. Security isn't the only reason why algorithms and protocols are selected. Where an algorithm, signature scheme, or a protocol isn't compatible with existing technology users will opt for the insecure route that still allows them to communicate. You may ask why a certain company uses RSA over DSA for their signature algorithm. It could be for security; it could also be for compatibility.

Recommendations

The **RSACng** class in .NET 4.6 offers great functionality that affords better security than older implementations. Our recommendation migrate from **RSACryptoServiceProvider** to **RSACng** where available, unless legacy compatibility is required. **RSACng** is also much easier to use with certificates (this will be covered in 11).

Table 29 contains the *minimum* key size recommendations for digital signature generation in modern secure environments as of 2016 (assuming security strength of 112-128 bits).

Table 29: Key size recommendations for digital signature algorithms

Algorithm	Key Size

RSA	2048 bits or larger
ECDSA	224 bits or larger (P256 or larger in .NET)
DSA	2048 bits or larger (*p*)

SHA256 is recommended for hashing in digital signature schemes. SHA1 should not be used in new production systems due to its weak collision resistance (less than 80 bits).

Pss is recommended for signature padding where it can be implemented rather than PKCS1. However, PKCS1 is widely used and will probably present legacy issues for a while.

Chapter Summary

- Digital signatures are a cryptographic primitive and use an asymmetric algorithm.
- Standardized digital signing processes usually call for a message's hash to be signed with a private key, producing the digital signature, which usually accompanies the message. Integrity of the message in question is ensured through the verification process using the public key and the accompanying signature.
- .NET offers a DSA implementation through the **DSACryptoServiceProvider** class and an elliptic curve variation of DSA through the **ECDsaCng** class.
- The **DSASignature[De]Formatter** class is recommended for further abstraction when using **DSACryptoServiceProvider**.
- **RSACryptoServiceProvider** and **RSACng** classes are used to sign data using an RSA key pair.
- Combing encryption and digital signing—signcryption—can be achieved through different orders of encryption and authentication. Encrypt-and-authenticate offers true nonrepudiation and allows for parallel encryption and signing.
- Signature schemes can be hardened using identifiers for the identity of parties and message sequence numbers.

Chapter Questions and Exercises

1. Explain the purpose of a digital signature and how it relates to the public key model.
2. Why is trust important with digital signing?
3. What additional steps need to be taken to use **DSACryptoServiceProvider** keys with **CspParameters**?
4. What are the differences between signing and verification methods in **RSACryptoServiceProvider** and **RSACng** classes?
5. Explain how the Horton principle relates to digital signatures and what problems identifiers can solve.
6. Explain the different orders of asymmetric encryption and digital signing (signcryption) and their relative strengths and weaknesses.

Scenarios

1. You are brought in on a meeting that was called to discuss the signature and encryption schemes being used in a secure messaging application. The stakeholders have decided on an encrypt-and-authenticate approach to secure and sign the data. Transport security will be provided through SSL and will not be a concern. But the data at rest still needs to be secure and maintain nonrepudiation, which the encrypt-and-authenticate signcryption scheme is expected to provide. A senior developer notes that the transport security and nature of the transport protocol will protect against replayed or retransmitted packets, rendering any message sequence numbers within the application unnecessary. What do you think about this? Is there still a reason to implement a message numbering system? What factors or variables might this depend on? Are there any other threats the developer may not have considered? Explain your position.

10 The Data Protection API (DPAPI)

Chapter Objectives

1. Understand the protections offered by the **ProtectedData** and **ProtectedMemory** classes.
2. Recognize the benefits and drawbacks of **ProtectedData** versus implementing a symmetric algorithm using a **SymmetricAlgorithm** subclass.
3. Learn how scope applies to the **ProtectedData** and **ProtectedMemory** classes.
4. Learn how to protect in-memory data or keys with **ProtectedMemory**.
5. Implement **ProtectedData** and **ProtectedMemory** in an example scenario.

Microsoft makes available native data protection services, partially, through the Data Protection API (DPAPI). DPAPI classes are available through the **System.Security.dll**. **ProtectedData** and **ProtectedMemory** are two popular classes that can be used to encrypt and decrypt data and memory. Static **Protect** and **Unprotect** methods of both classes provide the primary functionality. The scope of the protection provided by the DPAPI is controlled through the **DataProtectionScope** and **MemoryProtectionScope** enums.

Protected Data

The DPAPI's **ProtectedData** class is used to encrypt and decrypt data simply and easily without the developer having to interact with encryption algorithms or keys. The keys are managed by Windows and associated with either a current user or the local machine. This scope is specified using the **DataProtectionScope** enum (**DataProtectionScope.CurrentUser** or **DataProtectionScope.LocalMachine**).

The **Protect** and **Unprotect** methods are extremely easy to use. Three parameters are required: the data (byte array), additional entropy (byte array), and the **DataProtectionScope**). The additional entropy is not required and can be given a null value (we will go into different uses for this shortly).

Sensitive information can be protected using the **Protect** method like this:

```
byte[] secretBytes = ...

byte[] protectedBytes = ProtectedData.Protect(secretBytes, null,
DataProtectionScope.CurrentUser);
```

And unprotected just as easily:

```
byte[] unProtectedBytes = ProtectedData.Unprotect(protectedBytes, null,
DataProtectionScope.CurrentUser);
```

ProtectedData is very useful, but has a few specific implementation issues.

- **Portability:** Data protection scope is restricted to either the current user or the local machine. This has a tremendous impact on how the class can be used. Prospects of remote storage should deter developers from using this class as it creates too much of a dependency on the access of a given machine.
- **Access:** Notice that the particular data protection scopes would allow another application running under the same account or machine to attack the protected data. In other words, the confidentiality of protected data is purely illusory to *anything* else within the same protection scope. We will look at a fix to this next.

Preventing Intra-Scope Reads

The problem is that other processes running within the same protection scope can unprotect data previously secured with the **Protect** method. To prevent this we need to create our own scope of protection using a shared secret. Anything with access to the secret is now in our custom scope. We can do this by specifying the secret in the second parameter of the **Protect** method that is usually used for adding optional entropy. You've probably already realized that this shared secret will be subject to the same problems that plague the symmetric encryption model: secure distribution. After all, the additional entropy that we specify is essentially being used as a private key.

Given the context of how most applications are developed to use **ProtectedData**, the two most common methods of supplying the shared secret tend to be a hardcoded key in the application code (a bad idea), or a user supplied password (a better idea). Of course you are not limited to just these two options. But other options such as remote key stores, hardware key stores, certs, negotiated keys, what have you, usually aren't assumed to be in the context of solutions that consider **ProtectedData** to be their primary data protection mechanism. What this means is that if you have the resources to integrate remote key storage into your solution, you most likely have a context that allows better options than **ProtectedData**.

Assuming that your solution uses **ProtectedData** as part of a data security strategy, and needs a shared secret to limit the protection scope, getting the secret through user input is the best option. The user password can then be strengthened and stretched to the appropriate length using a key derivation protocol like PBKDF2. These methods are explained in detail in the chapter 5. A cryptographic hash algorithm could also be used.

Tampering: *Just because a process within a data protection scope cannot successfully decrypt the data does not mean that it cannot overwrite the data or tamper with it!*

Protected Memory

All of the good stuff ends up in memory. Plaintext messages, keys, passwords. Safeguards like encryption work great while nobody is viewing the data, but at some point we need to decrypt the data so we can see it, introducing keys or other sensitive data into memory. Once in memory we run the risk of an attacker getting access to the memory and reading it. There is also a risk of sensitive memory contents getting paged to a swap file and just sitting there on the hard disk. Some successful attacks have strategically forced sensitive memory contents to a swap file and then stolen the disk or its contents. Protecting this data in memory adds another layer of defense.

The **ProtectedMemory** class allows the developer to easily encrypt and decrypt in-memory data. This is useful for applications that handle secure information such as keys and passwords. Like the **ProtectedData** class, **ProtectedMemory**'s primary functionality is through the **Protect** and **Unprotect** methods. These methods, however, call for a different protection scope enum called **MemoryProtectionScope**, and do not take arguments for additional entropy. The data being protected by the memory must be of a 16-byte multiple.

Here's a quick example of how we could protect a byte array:

```
//Encryption key introduced into memory.
byte[] encryptionKey = new byte[16];

//Protect the encryption key in memory.
ProtectedMemory.Protect(encryptionKey, MemoryProtectionScope.SameProcess);
```

As we can see, right after the key is introduced, the memory is protected with a **SameProcess** protection scope. This means that only the code calling from the same process can unprotect the memory.

We unprotect the memory at the point which we want to use it. In the case of encryption code, we would protect the key as soon as it enters memory and unprotect it directly before encryption. After the encryption is complete we would want to protect the memory again even if we are intending to do some type of subsequent cleanup. Below we unprotect our key, presumably perform the encryption, and then protect the key memory again afterword.

```
//We want to use the key. Unprotect the memory.
ProtectedMemory.Unprotect(encryptionKey, MemoryProtectionScope.SameProcess);

//Perform the encryption...

//Protect the key again as soon as the encryption is complete.
ProtectedMemory.Protect(encryptionKey, MemoryProtectionScope.SameProcess);
```

The available memory protection scopes are narrow enough to restrict access to the same process, which is what we used in our examples, and what you'll want in most contexts. Still, you can also use the **CrossProcess** and **SameLogon** scopes. Table 30 describes the **MemoryProtectionScope** enum.

Table 30: MemoryProtectionScope Elements

MemoryProtectionScope Element	**Description**
SameProcess	The memory can only be unprotected by code running in the same process that protected it.
CrossProcess	The memory can be unprotected by any process.
SameLogon	The memory can be unprotected by any code running under the user.

DpapiDataProtector

The **DpapiDataProtector** class derives from **DataProtector** and performs simple data protection like the **ProtectedData** class we covered earlier this chapter. **DpapiDataProtector**, however, is newer and offers richer functionality while still taking care of the key management issue.

Creating an instance of **DpapiDataProtector** requires you to specify three strings in the class constructor: the application name, the primary purpose for the data protector, and a specific purpose of the data protector:

```
DpapiDataProtector protector = new DpapiDataProtector("application","primary purpose","spec
ific purpose");
```

The same **DataProtectionScope** used earlier this chapter can be supplied in the Scope property of the object. **CurrentUser** is the default scope. The **Protect** and **Unprotect** methods are used to encrypt and decrypt byte-array data:

```
byte[] data = new byte[32];

var protectedBytes = protector.Protect(data);

var unprotectedBytes = protector.Unprotect(protectedBytes);
```

Recommendations

For high-security applications the **ProtectedData** class should not be relied upon to secure sensitive information; especially where accounts/machines are not secured well. **ProtectedMemory** should be used to mitigate risks presented by sensitive in-memory data being exposed through swap files, memory dumps, or situations that might compromise memory contents. Because **ProtectedMemory** is simple to implement it poses a relatively low cost to developers given its added protections.

Chapter Summary

- The Data Protection API (DPAPI) gives developers an easy way to protect data and memory without the need to handle keys or secrets.
- **ProtectedData** can protect data within the scope of the local machine or a user account. Additional entropy can be defined during the operation. If kept secret, this can narrow the data protection scope to whatever possess this secret.
- **ProtectedMemory** can protect memory within the scope of the logon, the process, or between processes. This class can be very useful in building secure applications by protecting sensitive data that resides in memory.

Chapter Questions and Exercises

1. Explain the basic capabilities of the **ProtectedData** and **ProtectedMemory** classes. What are the differences between them?
2. Explain the different options using **DataProtectionScope**.

3. Write a program that uses the **ProtectedData** and **ProtectedMemory** classes to protect and unprotect data and memory.

Scenarios

1. You are developing a secure application. Stakeholders in the project express concerns that sensitive data in memory could be persisted to a swap file or exposed in a dump. What might you recommend to alleviate their concerns and reduce the risk of exposing sensitive contents in memory? How much additional labor might this entail?

11 Handling Certificates

Public-Key Certificate: *A set of data that uniquely identifies an entity, contains the entity's public key and possibly other information, and is digitally signed by a trusted party, thereby binding the public key to the entity. Additional information in the certificate could specify how the key is used and its cryptoperiod.*

Chapter Objectives

1. Understand certificate stores and certificate locations.
2. Understand how to access different certificate stores.
3. Learn how to obtain specific information from a certificate programmatically.
4. Learn how to use certificate keys to perform cryptographic operations.
5. Learn how to import and export certificates using the **X509Certificate2** class.

As discussed in the overview, a certificate provides information about someone's or something's identity; like a driver's license. X509v3 is the current standard for certificates. X509 certificates contain identity information like the issuer, public key, thumbprint, signature algorithm, dates of validity, and serial number. Certificates are used commonly for establishing trusted communications with servers over SSL, and digitally signing applications, code, and documents. This chapter will focus more on the basics of working with X509 certificates programmatically.

Certificates in Windows

Windows provides a pretty straightforward certificate management and storage system. Certificates can be imported, exported, and managed in a variety of formats. In terms of storing and accessing certificates in Windows, there are certificate stores and certificate store locations.

Store locations determine the scope of the access to the certificates. **Current User** and **Local Machine** are the main locations. **Current User** limits access to the current user account. **Local Machine** is broader and gives access to anyone on the machine itself.

Certificate stores are used to separate certificates, usually by usage or purpose. There are many certificate stores in Windows and some utilities or APIs will provide different levels of access. You will notice that they show different numbers of stores and different store names. We will cover programmatic store management in .NET (covered next). Manual store management is also available using the Certificate Management Console (MMC).

System.Security.Cryptography.X509Certificates

The **System.Security.Cryptography.X509Certificates** namespace is concerned with X509 certificates (as the name indicates) and is included in .NET. It offers basic certificate functionality such as opening certificates and retrieving certificate data. The namespace was expanded significantly in .NET 2 to provide certificate store access, updated certificate classes, and other enhancements. .NET 4.6 brought more functionality with certificate extensions that provide easier access to certificate keys.

At the nucleus of the namespace is the **X509Certificate2** class. This class contains several members for accessing various parts of a certificate, importing and exporting certificate material, and creating X509 certificates from files. **X509Certificate2** inherits from its older predecessor, **X509Certificate**. Our first example will show how to access an **X509Certificate2** using the **X509Certificate2Collection** and **X509Store** classes.

Before we get started make sure to reference the **System.Security.Cryptography.X509Certificates** namespace in your .cs file:

```
using System.Security.Cryptography.X509Certificates;
```

Opening a Certificate from a Store using X509Store

The first thing we need to do is create an instance of the **X509Store** class:

```
X509Store store = new X509Store();
```

The **X509Store** class will default to **My** for the store name and **CurrentUser** for the store location. We can also change either or both of these through the class's constructor using the **StoreName** or **StoreLocation** enums:

```
X509Store store = new X509Store(StoreName.Root,StoreLocation.CurrentUser);
```

Table 31 and Table 32 describe the **StoreLocation** and **StoreName** enums, respectively.

Table 31: StoreLocation Elements

StoreLocation Element	Description
CurrentUser	X509 store used by the current user.
LocalMachine	X509 store assigned to the local machine.

Table 32: StoreName Elements

StoreName Element	Description
AddressBook	X509 store for other users.
AuthRoot	X509 store for third-party CAs.
CertificateAuthority	X509 store for intermediate CAs.
Disallowed	X509 store for revoked certificates.
My	X509 store for personal certificates.
Root	X509 store for trusted root CAs.
TrustedPeople	X509 store for people or resources that are directly trusted.
TrustedPublisher	X509 store for publishers that are directly trusted.

Once we have instantiated an **X509Store** we need to open the store using the **Open(OpenFlags)** method. An **OpenFlags** enum must be specified to open the store and determines the type of store access. Table 33 contains the elements in the **OpenFlags** enum. Here, we're using **OpenFlags.OpenExistingOnly**, which will open a store if it exists, whereas other flags will create one if it does not:

```
store.Open(OpenFlags.OpenExistingOnly);
```

Table 33: OpenFlags Elements

OpenFlags Element	Description
IncludeArchived	Includes archived certificates when opening the store.
MaxAllowed	Opens the store for the highest access allowed.
OpenExistingOnly	Opens a store only if it exists.
ReadOnly	Opens with read only access.
ReadWrite	Opens with read/write access.

Once the store is open we can access its **X509Certificate2Collection** through the **Certificates** property. This collection contains all of the certificates from a particular store and presents them in instances of **X509Certificate2**:

```
X509Certificate2Collection certs = store.Certificates;
```

We can access a certificate through its indexer or by enumerating through the collection to find a particular certificate or certificate value. Below we access the first certificate in the collection. However, enumerating a collection will be a more effective means of obtaining a certificate based on specific criteria.

```
X509Certificate2 cert = certs[0];
```

Since we've obtained the certificate collection and we don't need any other interaction with the store we can close it using **Close()**:

```
store.Close();
```

Now we can use the **X509Certificate2** members to obtain any information we want about the certificate itself. These include for example **FriendlyName**, **Issuer**, **NotAfter**, **and SerialNumber**. We'll try out a few of these with our certificate:

```
string friendlyName = cert.FriendlyName;

string issuerCA = cert.Issuer;

string expireDate = cert.NotAfter.ToString();

string serial = cert.SerialNumber;
```

That's it. Here's everything we just did:

```
X509Store store = new X509Store();

store.Open(OpenFlags.OpenExistingOnly);

X509Certificate2Collection certs = store.Certificates;

store.Close();

X509Certificate2 cert = certs[0];

string friendlyName = cert.FriendlyName;

string issuerCA = cert.Issuer;

string expireDate = cert.NotAfter.ToString();

string serial = cert.SerialNumber;
```

Example: Finding Certificates using X509FindType

In the last example we obtained a certificate from a certificate store by specifying the zero index in a collection. This isn't acceptable where you have to pull a certificate from a collection based on criteria. Where you need to obtain a certificate from a store collection using information that relates to the certificate itself, such as issuer name, serial number, or usage, **X509FindType** can help.

The **X509Certificate2Collection** class has a **Find** method that queries the collection using an **X509FindType** enum, which specifies the certificate property to look at, and an object value that is used to compare to the property referenced by the enum. Let's try this out.

Below we get an instance of the certificate store in an **X509Certificate2Collection** object.

```
X509Certificate2Collection storeCollection = store.Certificates;
```

Let's say that we want to find any certificates in this store where the issuer name is, or contains, the string "localhost". Using the **Find** method in the **X509Certificate2Collection** class, we will supply the

X509FindType.FindByIssuerName enum and a string to match (case insensitive) the issuer name, in this case "localhost"; the third parameter calls for a bool value indicating whether only valid certificates should be returned:

```
X509Certificate2Collection localCerts = storeCollection.Find(X509FindType.FindByIssuerName,
"localhost", false);
```

We could also go to the root store and grab a collection of certificates issued by "Verisign":

```
X509Store store = new X509Store(StoreName.Root);

store.Open(OpenFlags.OpenExistingOnly);

X509Certificate2Collection storeCollection = store.Certificates;

store.Close();


X509Certificate2Collection verisignCerts =
storeCollection.Find(X509FindType.FindByIssuerName, "Verisign", true);
```

You should keep in mind that not all of the X509FindType elements search using the same criteria. X509FindType.FindByIssuerName, for example, is case insensitive and will find anything that contains or matches the supplied string. In contrast, X509FindType.FindByIssuerDistinguishedName is case sensitive and will look for an exact string match, thus providing higher precision than the FindByIssuerName criteria.

There is a find type element to search practically every field on a certificate. We will not cover the rest of them here, but you should be aware that they exist and can be very helpful.

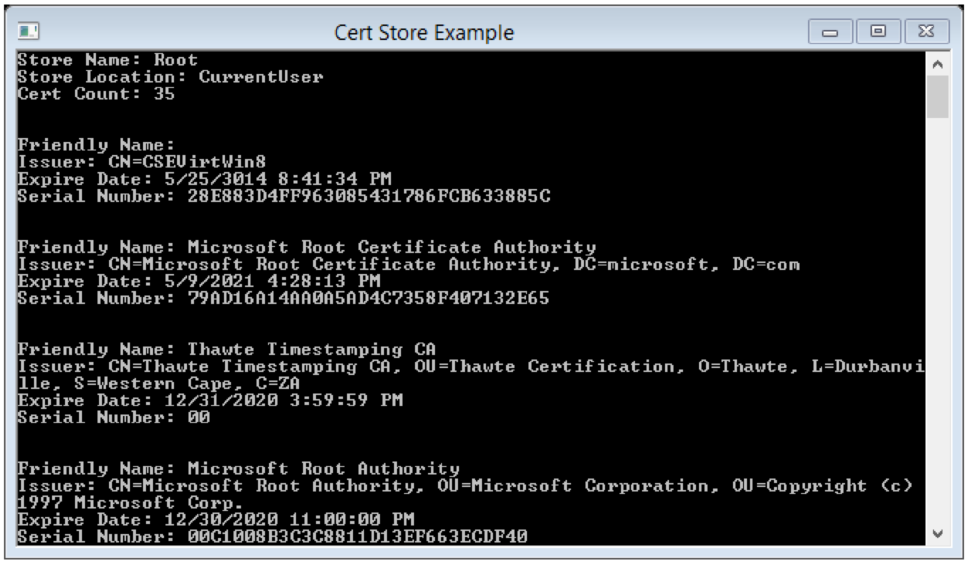
Example: Listing a Store

In this console app example we will list all of the certificates in the root store, providing a little information on each using the tools from last section. Notice that in this example, instead of letting our store default to My, we're setting our store to Root, which contains root CA certificates.

```
X509Store store = new X509Store(StoreName.Root);
store.Open(OpenFlags.OpenExistingOnly);

X509Certificate2Collection certs = store.Certificates;

store.Close();

Console.WriteLine("Store Name: " + store.Name);
Console.WriteLine("Store Location: " + store.Location);
Console.WriteLine("Cert Count: " + certs.Count.ToString() + "\n\n");

foreach (var cert in certs)
{
    Console.WriteLine("Friendly Name: " + cert.FriendlyName);
    Console.WriteLine("Issuer: " + cert.Issuer);
    Console.WriteLine("Expire Date: " + cert.NotAfter);
    Console.WriteLine("Serial Number: " + cert.SerialNumber + "\n\n");
}

Console.ReadKey();
```

Figure 46 shows the store being listed.

Figure 46: Listing the Store



```
Store Name: Root
Store Location: CurrentUser
Cert Count: 35


Friendly Name:
Issuer: CN=CSEVirtWin8
Expire Date: 5/25/3014 8:41:34 PM
Serial Number: 28E883D4FF963085431786FCB633885C


Friendly Name: Microsoft Root Certificate Authority
Issuer: CN=Microsoft Root Certificate Authority, DC=microsoft, DC=com
Expire Date: 5/9/2021 4:28:13 PM
Serial Number: 79AD16A14AA0A5AD4C7358F407132E65


Friendly Name: Thawte Timestamping CA
Issuer: CN=Thawte Timestamping CA, OU=Thawte Certification, O=Thawte, L=Durbanvi
lle, S=Western Cape, C=ZA
Expire Date: 12/31/2020 3:59:59 PM
Serial Number: 00


Friendly Name: Microsoft Root Authority
Issuer: CN=Microsoft Root Authority, OU=Microsoft Corporation, OU=Copyright (c)
1997 Microsoft Corp.
Expire Date: 12/30/2020 11:00:00 PM
Serial Number: 00C1008B3C3C8811D13EF663ECDF40
```

Example: Accessing Keys in a Certificate before .NET 4.6

Our first step will be to obtain the cert as an X509Certificate2. We simply use our first cert from the My store, but you can substitute this with any other cert that has an RSA key since you might not have any certificates in My store (you could also try StoreName.My plus StoreLocation.LocalMachine in the store constructor):

```
X509Store store = new X509Store(StoreName.My);

store.Open(OpenFlags.OpenExistingOnly);

X509Certificate2 cert = store.Certificates[0];

store.Close();
```

X509 certificates contain an asymmetric public key. This allows us to take a publicly sharable cert (and public key) and encrypt data that can only be decrypted using the private key, which is kept safe by the cert owner. The X509Certificate2 class contains a PublicKey and PrivateKey property that allows you to access the keys associated with the cert. The HasPrivateKey property can be used to ascertain whether the cert has a private key. Only your certs will have a private key that you can access. Only RSA and DSA keys are supported by the X509Certificate2 class and it will throw an exception if the either key is not supported. Table 34 contains the members of PublicKey.

The HasPrivateKey property in X509Certificate2 is used to determine if a certificate has a private key. This should be used in regular practice to avoid trying to access a private key that doesn't exist:

```
if (cert.HasPrivateKey)
{

}
```

Table 34: PublicKey members

PublicKey Member	Description
EncodedKeyValue	The ASN.1 encoded key value.
EncodedParameters	The ASN.1 encoded public key parameters.
Key	The **AsymmetricAlgorithm** object representing the public key.
Oid	The object identifier (OID) of the public key.

Similar to what we covered in chapter 8, you can also import and export the **AsymmetricAlgorithm** object representing the public key as XML:

```
string keyXml = cert.PublicKey.Key.ToXmlString(false);
```

Importing the XML into an **RSACryptoServiceProvider** object is just as easy:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

rsa.FromXmlString(keyXml);
```

Example: Accessing Certificate Keys in .NET 4.6 with RSACertificateExtensions

Last section showed how to access keys through the **PublicKey** and **PrivateKey** properties in X509Certificate2. .NET 4.6 makes accessing certificate keys much easier using the **RSACertificateExtensions** extension methods. These methods, **GetRSAPublicKey** and **GetRSAPrivateKey**, return an **RSA** object (**RSACng** instance) containing the key.

Assuming we have an **X509Certificate2** instance (represented below as **cert**) we can use **RSACertificateExtensions** like this:

```
RSA publicKey = RSACertificateExtensions.GetRSAPublicKey(cert);

RSA privateKey = RSACertificateExtensions.GetRSAPrivateKey(cert);
```

This approach is recommended if you have access to .NET 4.6 and are not facing any legacy compatibility issues. Certificate extensions are also available for **ECDsa** and are used in the same fashion as shown above.

Encryption and Decryption with Certificates

Encryption or decryption can be performed in typical fashion once the necessary object/key has been obtained from the cert. Refer to chapter 8 for examples. Remember, you will only have access to the public key unless you are handling your own certs in a format that exposes the private key.

Next we encrypt some byte-array data with the **RSACng** instance returned from the **RSACertificateExtensions** class, then decrypt it with the private key. We are specifying **OaepSHA1** padding:

```
RSA publicKey = RSACertificateExtensions.GetRSAPublicKey(cert);

RSA privateKey = RSACertificateExtensions.GetRSAPrivateKey(cert);
```

```
byte[] data = new byte[32];

var encryptedData = publicKey.Encrypt(data, RSAEncryptionPadding.OaepSHA1);

var plaintext = privateKey.Decrypt(encryptedData, RSAEncryptionPadding.OaepSHA1);
```

If you are using a version of .NET older than .NET 4.6 you will have to obtain your keys/algorithms as shown on page 184.

Programmatically Importing and Exporting Certificates

The **X509Certificate2** class allows developers to import and export certificate material. Importing occurs through either the class constructor or the **Import** method. There are different overloads that allow you to specify file names, passwords (if it is password protected), or different formats for certs. Here's a quick example of importing a .pfx (Personal Information Exchange) cert file located for us in *C:\cert.pfx* that is password protected and contains a private key. The filename is the first argument and the cert password is the second (a secure string can also be used instead of a regular string):

```
var cert = new X509Certificate2("C:/cert.pfx", "yourcertpassword");
```

The **Export** method will export a cert as a byte array. In this process you're able to select the export format using the **X509ContentType** enum, and specify a password (optional). **X509ContentType.Pkcs12** will function the same as **X509ContentType.Pfx** and export a private key with the cert if it's allowed. **X509ContentType.Cert** (.cer) is another common type and will only export a public key. Below we export a cert from a user store in .pfx format, which will also export its private key and all the certificates in its path:

```
X509Store store = new X509Store();

store.Open(OpenFlags.OpenExistingOnly);

X509Certificate2 cert = store.Certificates[0];

store.Close();

byte[] exportedCert=cert.Export(X509ContentType.Pfx,"somepassword");
```

X509ContentType elements:

- Authenticode
- Cert
- Pfx
- Pkcs12
- Pkcs7
- SerializedCert
- SerializedStore
- Unknown

We could also export the same cert as a .cer file using **X509ContentType.Cert**. This will only export the public key and will not include other certificates in the path:

```
byte[] exportedCert=cert.Export(X509ContentType.Cert);
```

Once we have the exported byte array we can use it as we please. Below we'll write the .pfx cert to a file:

```
using (var fs = new FileStream("C:/exportedCert.pfx", FileMode.Create))
{
    fs.Write(exportedCert, 0, exportedCert.Length);
    fs.Close();
}
```

Web Service Enhancements (WSE)

In terms of cryptography, the Web Services Enhancements (WSE) for .NET have traditionally afforded better options for working with certificates and accessing system certificate stores than the **System.Security.Cryptography.X509 namespace**. .NET 2.0, however, brought some additional features for handling certificates, and .NET 4.6 has introduced better even functionality (we covered the new certificate extensions objects for easier key access earlier this chapter).

WSE is available for download (at the time of this writing): http://msdn/webservices/building/wse

CAPICOM

CAPICOM is the COM wrapper for the Microsoft Cryptography API (Crypto API or CAPI). CAPICOM can be used to provide similar data protection services to those in the **System.Security.Cryptography** namespace but offers finer grained approaches to public key tasks and provides better access to key stores. CAPICOM does not come with .NET and will have to be installed separately.

Recommendations

Certificates are used in conjunction with standardized cryptographic protocols. Wherever possible, these protocols, like SSL for instance, should be used over custom solutions. Additionally, certificate usage and management should be accomplished as much as possible through existing means. For instance, where IIS allows you to import and specify a certificate, opt for this rather than trying to program your own version or solution. Productivity and security are the primary drivers for this, but formatting and maintenance hassles are close secondaries.

RSACng is recommended for using certificate keys for encryption or signing because it works with **RSACertificateExtensions** in .NET 4.6 and provide a much cleaner approach than older methods.

Chapter Summary

- Certificate access in .NET is achieved primarily through the **System.Security.Cryptography.X509Certificates** library. Other resources, such as WSE, CAPICOM, and third-party libraries provide additional functionality.
- Certificate management can be handled through the MMC console or programmatically using certificate locations and stores.
- The **X509Certificate2** class allows import and export of certificate material and access to certificate information.

- The **X509Store** class is used to control access to different stores and store locations. **X509Certificate2Collection** objects contain a collection of **X509Certificate2** instances and allows basic queries on the collection.
- .NET 4.6 has introduced certificate extensions objects that allow easier access to certificate keys. Older versions of .NET will have to access keys through the **PublicKey** and **PrivateKey** properties of the **X509Certificate2** class.

Chapter Questions and Exercises

1. Explain how certificate stores and store locations work.
2. Use the MMC console to browse your own stores.
3. Describe the capabilities of the **X509Certificate2** class.
4. Write an application that has to find a specific certificate in a certificate store using the **Find** method and **X509FindType** enum in the **X509Certificate2Collection** class.
5. Perform an import and export of a certificate manually with MMC Certificate console as well as programmatically.

12 Best Practices for Testing and Development

Complexity is the Enemy of Security

Complexity is the enemy of security. This point was heavily emphasized in Schneier's *Secrets and Lies*. Security in general is best achieved through simple interfaces and processes. Features should be minimized, especially by default. Each decision point—each contingency presented in a new feature—increases your attack surface (the area capable of being attacked), making your application harder to secure. Complex systems on average tote a much larger attack surface than simpler ones. Ultimately, the most secure systems are often the most Spartan.

Exception Handling and Logging

The first point that needs to be made here, even though we covered it elsewhere in the book, is that exceptions contain some of the most helpful information about what is going on inside an application when it fails. Exceptions give attackers access to the state of our applications and the data they contain. You have to catch exceptions that otherwise might make their way to the surface and leak sensitive data. This can be accomplished at a lower level by controlling how exceptions are handled using try/catch blocks. At a higher level, depending on the environment, release/deployment options are often configurable to offer a safe error screen for customers while providing more detailed error information for developers. Exceptions are commonly logged to give developers better information on how to fix issues. Developers can often provide more effective and faster solutions where they have more detailed logging. But again, this information must be secured from the prying eyes of attackers.

Logging also can notify you of an attack but too often is left ignored and overlooked, or improperly assessed by automated systems like intrusion prevention systems (IPS) that have cried wolf.

No Secrets in Code or Configuration Files

This might be the most important and widely broken rule in secure development. If you hardcode a password or key into application code, you have just destroyed your security because the only mechanism used to secure your data is publicly accessible. Whether it's an oversight or a tactic relying on obscurity, it's going to be game over. Often this is the result of poor planning for key management, which is covered next. Configuration files are also commonly susceptible to attack and often contain secrets such as plaintext database connection strings, keys and passwords.

Plan for Key Management

Planning for key management is critical on bigger projects and usually involves coordinating with a systems or security administrator. Developing a key management solution is outside the scope of this book and depends heavily on the application.

NIST is a good resource for learning about key management and good policy practices. The meat of their key management recommendations are contained in NIST Special Publication 800-57, parts 1-3, and Special Publication 800-130. They also have a key management project.

Reduce Sensitive Information in Memory

Reducing sensitive information in memory is just good practice. If you have a byte array in memory that contains an encryption key, try not to make 40 copies of it throughout the course of the program. If you keep an eye on memory you'll also notice that it'll help increase simplicity and add to better designed solutions. Sensitive information in memory can be further reduced by protecting (encrypting it) using the **ProtectedMemory** class covered on page 175.

FXCop

FXCop is a code analyzer that should be run on your project as part of the regular development and testing process. FXCop will catch many issues developers don't notice or in many cases don't even fully understand. Using this as part of an iterative development cycle is recommended as some of the issues it may catch could entail refactoring, changing the interfaces you expose or consume.

Sane and Secure Defaults

Too many developers treat security like a feature that should be added on or enabled, making their applications insecure by default. Instead, the development process should integrate security from design to deployment:

- Build security from the ground up, incorporating secure design methodologies. Here solutions are designed for security, not simply made secure through some "add on".
- The product should be secure out of the box and employ secure default settings and configurations. Users should be able to easily use the product in a secure manner without additional configuration.
- Applications need to be securely installed, administered, and updated. Things to think about here are:

- How will the system stay secure?
- Will it need updates or patches?
- How can it be serviced or updated in a secure manner?
- Can administrators securely manage the system?

Least Privilege

The principle of least privilege is an axiomatic element in practically every secure environment or application. Least privilege means that you're giving the user, the process, the program—whatever is executing—the least amount of privilege necessary to do what it needs to.

Defense in Depth

Defense in depth is another axiomatic security principle. Multiple security measures that provide a layered approach are obviously more secure than a single security measure. Defense in depth increases security by giving the attacker multiple obstacles to overcome in terms of what they need to compromise.

A datacenter, for example, houses secure assets. To keep them safe there will be multiple layers of security: Fences, cameras, guards, access cards, passcodes, mantraps, motion sensors, and door locks. And this is only considering physical access to resources. Imagine the list if we add all of the logical access control components.

Defense in depth should be used whenever possible to increase security and restrict access to assets. For secure applications, this can include additional privacy controls, digital signing, salts, strong passwords/keys, and additional help from the OS in terms of ACLs and account access.

Cryptography in Mobile

The number of applications developed for mobile devices has exploded. Part of this has been aided by cross-platform development. Cross-platform tools save time because programmers can develop against familiar APIs without dealing with the details of the underlying device or OS. This equals more apps in less time with fewer implementation issues. One of the challenges with this type of development is accounting for the security models associated with various devices.

Mobile devices such as phones have notoriously poor security. Successful attacks have capitalized on exploits in native security models, programming errors, and the trend that user access to resources and sensitive data has been relaxed in favor of customer satisfaction (until there is a data breach).

We have to look at where things are actually being stored given the platform, the native security of the platform, and what access levels the attackers have. We are not going to get into device/platform-specific issues, but you should thoroughly research the devices for which you are developing, their security issues, and what steps need to be taken to obtain the security level you need.

With mobile device security you must always assume the attacker will gain physical access. Other assumptions should be made as well:

- Security is generally weaker.

- Users will use a weak numeric PIN or password to protect access. This often includes access to networked resources.
- The attacker will pull all data locally stored on the device.
- Mobile libraries may not contain a rich assortment of cryptographic algorithms and tools, especially those that are taking on resources.
- Due to weaker I/O and unreliable network access many developers will prefer to perform cryptographic operations on the device and store data and keys locally.

Keeping Up on Research

Cryptography is one of those fields that requires you to stay current on your research. The security of most of the common algorithms has been very well tested. The RSA algorithm, for instance, has been publicly tested for decades, but is still widely used and trusted. Other algorithms, due to advances in computing technology or recent security developments have been deemed no longer suitable for production systems.

It's important that before you implement an algorithm, or set of algorithms, in an application, that you determine if they are currently considered secure and if they are recommended for retirement in the near future. Many production systems are built with the expectation that they will be used for 10-20 years. This means their programmers shouldn't use an algorithm that is recommended to be transitioned out over the next 5 years.

To stay current on these types of recommendations, the National Institute of Standards and Technology (NIST) is a great resource. NIST provides in-depth research on standardized algorithms, their lifespans, and those that are currently being reviewed.

Additional Resources

We recommend *Writing Secure Code 2nd Ed.* as an additional resource. It is rich with valuable information on secure development and testing practices. It covers threat modeling, design, Windows specific issues, common cryptographic issues, and also has sections on securing .NET.

Glossary

For accuracy and standardization, many of these definitions have been obtained from the National Institute of Standards and Technology.

$\oplus$	Bit-wise exclusive-or. A mathematical operation that is defined as: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, and $1 \oplus 1 = 0$.
&	Bit-wise AND. A mathematical operation for which the result is 1 if the first bit is 1 and the second bit is 1. Otherwise, the result is 0. That is, 0 & 0 = 0, 0 & 1 = 0, 1 & 0 = 0, and 1 & 1 = 1.
||	Concatenation
0xa	a is represented as a hexadecimal value.
0x00	An all-zero octet.
F(x)	A mathematical function with x as the input.
H(x)	A hash function with x as an input.
T(x, k)	Truncation of the bit string x to k bits.
2TDEA	Two-key Triple Data Encryption Algorithm
3TDEA	Three-key Triple Data Encryption Algorithm
ACL	Access Control List

AES	Advanced Encryption Standard
Access control	Restricts access to resources to only privileged entities.
Accountability	A property that ensures that the actions of an entity may be traced uniquely to that entity.
Algorithm Transition	The processes and procedures used to replace one cryptographic algorithm with another.
Approved	FIPS-**approved** and/or NIST-recommended. An algorithm or technique that is either 1) specified in a FIPS or NIST Recommendation, or 2) specified elsewhere and adopted by reference in a FIPS or NIST Recommendation.
Asymmetric key	A cryptographic key used with an asymmetric key (public key) algorithm. The key may be a private key or a public key.
Asymmetric key algorithm	A cryptographic algorithm that uses two related keys, a public key and a private key. The two keys have the property that determining the private key from the public key is computationally infeasible. Also known as a public key algorithm.
Authentication	A process that establishes the source of information, provides assurance of an entity’s identity or provides assurance of the integrity of communications sessions, messages, documents or stored data.
Authorization	Access privileges that are granted to an entity; conveying an “official” sanction to perform a security function or activity.
Bit	A binary digit: 0 or 1.
Bit Error	The substitution of a ‘0’ bit for a ‘1’ bit, or vice versa.
Bit String	An ordered sequence of 0’s and 1’s.
Birthday Attack	A type of collision attack. This concept is most often applied generically to hash functions, where an N-bit

	hash function will generate a collision after approximately $2^{n/2}$ different inputs. However, the birthday attack can be used to determine the likelihood of a collision in any finite space using the same formula—about √N, or $2^{n/2}$, where N is the bit size of the space.
Block	A bit string whose length is the block size of the block cipher algorithm.
Block Cipher	A family of functions and their inverse functions that is parameterized by cryptographic keys; the functions map bit strings of a fixed length to bit strings of the same length.
Block Size	The number of bits in an input (or output) block of the block cipher.
CA	Certificate (Certification) Authority
CBC	Cipher Block Chaining.
CBC-MAC	Cipher Block Chaining-Message Authentication Code
CCM	Counter with Cipher Block Chaining-Message Authentication Code
CFB	Cipher Feedback.
CMS	Certificate Management System
CTR	Counter.
Certificate	See public-key certificate.
Certificate authority (CA)	The entity in a Public Key Infrastructure (PKI) that is responsible for issuing certificates and exacting compliance to a PKI policy.
Chosen-Plaintext Attack (CPA)	Almost identical to a known-plaintext attack, but in this model an attacker knows the corresponding ciphertext to a plaintext of their choosing. Rather than having to

	capture plaintext/ciphertext pairs the attacker can generate ciphertexts for whatever data they want.
Chosen-Ciphertext Attack (CCA)	An attack where the attacker has the ability to encrypt plaintexts of their choice as well as decrypt ciphertexts of their choice. This model gives the attacker a great deal of flexibility over the other models and a better chance at recovering a key.
Ciphertext	Encrypted data.
Ciphertext-Only Attack	An attack in which the attacker only has access to the ciphertext.
Collision	Two or more distinct inputs produce the same output.
Collision Attack	An attack that exploits the probability of finding a collision in data within a finite space.
Collision resistance	1. It is computationally infeasible to find two different inputs to a cryptographic function that have the same output. 2. It is computationally infeasible to find two different inputs to the cryptographic hash function that have the same hash value. That is, if *hash* is a cryptographic hash function, it is computationally infeasible to find two different inputs x and x' for which $hash(x) = hash(x')$. Collision resistance is measured by the amount of work that would be needed to find a collision for a cryptographic hash function with high probability. If the amount of work is 2^N, then the collision resistance is N bits. The estimated strength for collision resistance provided by a hash function is half the length of the hash value produced by a given cryptographic hash function, i.e., the estimated security strength for collision resistance is $L/2$ bits. For example, SHA-256 produces a (full-length) hash value of 256 bits; SHA-256 provides an estimated collision resistance of 128 bits.
Commercial Off-The-Shelf (COTS)	Technology and/or a product that is ready-made and available for sale, lease, or license to the general public.
Compromise	The unauthorized disclosure, modification or use of sensitive data (e.g., keying material and other security-related information).

Confidentiality Mode	A mode that is used to encipher plaintext and decipher ciphertext. The confidentiality modes in this recommendation are the ECB, CBC, CFB, OFB, and CTR modes.
Cryptanalysis	1. Operations performed in defeating cryptographic protection without an initial knowledge of the key employed in providing the protection. 2. The study of mathematical techniques for attempting to defeat cryptographic techniques and information system security. This includes the process of looking for errors or weaknesses in the implementation of an algorithm or in the algorithm itself.
Cryptographic algorithm	A well-defined computational procedure that takes variable inputs, often including a cryptographic key, and produces an output.
Cryptographic Boundary	An explicitly-defined perimeter that establishes the boundary of all components of a cryptographic module.
Cryptographic Hash Function	See Hash Function.
Cryptographic Key	A parameter used in the block cipher algorithm that determines the forward cipher operation and the inverse cipher operation.
Cryptographic key(key)	A parameter used in conjunction with a cryptographic algorithm that determines its operation in such a way that an entity with knowledge of the key can reproduce or reverse the operation, while an entity without knowledge of the key cannot. Examples include: 1. The transformation of plaintext data into ciphertext data, 2. The transformation of ciphertext data into plaintext data, 3. The computation of a digital signature from data, 4. The verification of a digital signature, 5. The computation of an authentication code from data, 6. The verification of an authentication code from data and a received authentication code, 7. The computation of a shared secret that is used to derive keying material. 8. The derivation of additional keying material from a key-derivation key (i.e., a pre-shared key).

Cryptographic Key Management System	A system for the management (e.g., generation, distribution, storage, backup, archive, recovery, use, revocation, and destruction) of cryptographic keys and their metadata.
Cryptographic module	The set of hardware, software, and/or firmware that implements security functions (including cryptographic algorithms and key generation) and is contained within a cryptographic boundary.
Cryptography	The use of mathematical techniques to provide security services, such as confidentiality, data integrity, entity authentication, and data origin authentication.
Cryptoperiod	The time span during which a specific key is authorized for use or in which the keys for a given system or application may remain in effect.
DSA	Digital Signature Algorithm specified in [FIPS186]
Data integrity	A property whereby data has not been altered in an unauthorized manner since it was created, transmitted or stored.
Data Block (Block)	A sequence of bits whose length is the block size of the block cipher.
Data Segment (Segment)	In the CFB mode, a sequence of bits whose length is a parameter that does not exceed the block size.
Decryption (Deciphering)	The process of a confidentiality mode that transforms encrypted data into the original usable data.
Decryption Oracle	Attackers can query a decryption oracle for a plaintext using a ciphertext. The chosen-ciphertext attack is an example of an attack that uses a decryption oracle.
Denial of Service (DOS) or Distributed Denial of Service (DDOS) Attack	Attacks intend to disrupt the availability of a system.

Deterministic random bit generator (DRBG)	An algorithm that produces a sequence of bits that are uniquely determined from an initial value called a seed. The output of the DRBG “appears” to be random, i.e., the output is statistically indistinguishable from random values. A cryptographic DRBG has the additional property that the output is unpredictable, given that the seed is not known. A DRBG is sometimes also called a Pseudo Random Number Generator (PRNG) or a deterministic random number generator.
Digital signature	The result of a cryptographic transformation of data that, when properly implemented, provides origin authentication, assurance of data integrity and signatory non-repudiation.
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECB	Electronic Codebook.
ECDSA	Elliptic Curve Digital Signature Algorithm specified in [ANSX9.62]
Encryption (Enciphering)	The process of changing plaintext into ciphertext using a cryptographic algorithm and key.
Encryption Oracle	An encryption oracle gives the attacker the ability to encrypt plaintexts of their choice and receive the ciphertext in return. Here the attacker uses a plaintext to query the oracle, which encrypts the plaintext using an unknown key, and exposes the ciphertext.
Entity	An individual (person), organization, device or process. An entity has an identifier to which it may be associated.
Entropy	The entropy of a random variable *X* is a mathematical measure of the expected amount of information provided by an observation of *X*. As such, entropy is always relative to an observer and his or her knowledge prior to an observation.

Ephemeral key	A cryptographic key that is generated for each execution of a key establishment process and that meets other requirements of the key type (e.g., unique to each message or session). In some cases, ephemeral keys are used more than once within a single session (e.g., broadcast applications) where the sender generates only one ephemeral key pair per message, and the private key is combined separately with each recipient's public key.
Exclusive-OR (XOR)	The bitwise addition, modulo 2, of two bit strings of equal length.
Exhaustive Search Attack	An attack that tries all possible values within a target space
FFC	Finite Field Cryptography
FIPS	Federal Information Processing Standard
Formatting Function	The function that transforms the payload, associated data, and nonce into a sequence of complete blocks.
Forward Cipher Function	One of the two functions of the block cipher algorithm that is determined by the choice of a cryptographic key.
Hardening	A process to eliminate a means of attack by patching vulnerabilities and turning off nonessential services. Hardening a computer involves several steps to form layers of protection.
Hash function	A function that maps a bit string of arbitrary length to a fixed-length bit string. Approved hash functions satisfy the following properties: 1. (One-way) It is computationally infeasible to find any input that maps to any pre-specified output, and 2. (Collision resistant) It is computationally infeasible to find any two distinct inputs that map to the same output.
Hash value	The fixed-length bit string produced by a hash function
HMAC	Hashed Message Authentication Code. Also, Keyed-Hash Message Authentication Code

IEEE	Institute of Electrical and Electronics Engineers
IFC	Integer factorization cryptography
IKE	Internet Key Exchange
IPSec	Internet Protocol Security
ISO/IEC	International Organization for Standardization/International Electrotechnical Commission
Identifier	A bit string that is associated with a person, device or organization. It may be an identifying name, or may be something more abstract (for example, a string consisting of an IP address and timestamp), depending on the application.
Identity	The distinguishing character or personality of an entity.
Initialization vector (IV)	A vector used in defining the starting point of a cryptographic process.
Inverse Cipher Function	The inverse function of the forward cipher function for a given cryptographic key.
Kerckhoff’s Principle	Assumes that an attacker knows the underlying details of an algorithm, such as IVs, ciphermodes, salts, etc., but not the secret input (usually a key).
Key agreement	A key establishment procedure where the resultant keying material is a function of information contributed by two or more participants, so that no party can predetermine the value of the keying material independent of the other party’s contribution.
Key derivation	1. A process by which one or more keys are derived from a shared secret and other information during a key agreement transaction. 2. A process that derives new keying material from a key that is currently available.

Key-derivation function	A function that, with the input of a cryptographic key or shared secret, and possibly other data, generates a binary string, called keying material.
Key generation	The process of generating keys for cryptography.
Key length	Used interchangeably with “Key size”.
Key management	The activities involving the handling of cryptographic keys and other related security parameters (e.g., passwords) during the entire lifecycle of the keys, including their generation, storage, establishment, entry and output, use and destruction.
Key Owner	An entity (e.g., person, group, organization, device, or module) authorized to use a cryptographic key or key pair.
Key pair	A private key and its corresponding public key; a key pair is used with an asymmetric key (public key) algorithm.
Key size	The length of a key in bits; used interchangeably with “Key length”.
Key wrapping	A method of encrypting and decrypting keys and (possibly) associated data using a symmetric key; both confidentiality and integrity protection are provided.
Key transport	A key-establishment procedure whereby one party (the sender) selects and encrypts the keying material and then distributes the material to another party (the receiver). When used in conjunction with a public-key (asymmetric) algorithm, the keying material is encrypted using the public key of the receiver and subsequently decrypted using the private key of the receiver. When used in conjunction with a symmetric algorithm, the keying material is encrypted with a key-encrypting key shared by the two parties.

Key update	A function performed on a cryptographic key in order to compute a new, but related key for the same purpose.
Known-Plaintext Attack	An attack in which the attacker knows the corresponding ciphertext of a plaintext encrypted under a particular unknown key. In this model the attacker tries to capture pairs of ciphertexts and their *known* plaintexts. The attacker looks for some type of discernable pattern or leakage that could give them info about the underlying plaintexts for other ciphertexts they intercept or the keys used to encrypt them.
Least Significant Bit(s)	The right-most bit(s) of a bit string.
MAC	Message Authentication Code.
Malware	Software designed and operated by an adversary to violate the security of a computer (includes spyware, virus programs, root kits, and Trojan horses)
Man-in-the-Middle Attack	A generic type of networked attack. In an MITM attack the attacker inserts himself or herself between a legitimate user and a target resource such as a server or another user.
Metadata	Information used to describe specific characteristics, constraints, acceptable uses and parameters of another data item (e.g., a cryptographic key).
Meet-in-the-Middle (MITM) Attack	A type of collision attack.
Mode of Operation	A set of rules for operating on data with a cryptographic algorithm and a key; often includes feeding all or part of the output of the algorithm back into the input of the next iteration of the algorithm, either with or without additional data being processed. Examples are: Cipher Feedback, Output Feedback, and Cipher Block Chaining.

Most Significant Bit(s)	The left-most bit(s) of a bit string.
NIST	National Institute of Standards and Technology
Non-repudiation	A service that may be afforded by the appropriate application of a digital signature. The signature provides assurance of the integrity of the signed data in such a way that the signature can be verified by any party in possession of the claimed signatory's public key. The assumption is that the claimed signatory had knowledge of the data that was signed and is the only entity in possession of the private key associated with that public key; thus, verification of the signature provides assurance to a verifier that the data in question was knowingly signed by none other than the claimed signatory.
Nonce	A time-varying value that has at most a negligible chance of repeating—for example, a random value that is generated anew for each use, a timestamp, a sequence number, or some combination of these.
OAEP	Optimal Asymmetric Encryption Padding
OFB	Output Feed Back
OID	Object Identifier
Octet	A string of eight bits.
Octet Length	The number of octets in an octet string.
Octet String	An ordered sequence of octets.
Oracle	A generic term that refers to something an attacker is able to query for information. Oracles are commonly used to gain information about plaintext data or encryption keys.

Origin authentication	A process that provides assurance of the origin of information (e.g., by providing assurance of the originator's identity).
Padding	Material that is used to make a plaintext a perfect length for an encryption process that requires fixed-length data. Padding must be attached in a way so that it can be identified and properly removed in the decryption process in a way that does not damage the original plaintext.
Padding Oracle	A padding oracle is a type of decryption oracle. However, instead of querying the oracle for a plaintext value, the oracle is only queried for an indication of whether or not the decryption succeeded. Block cipher modes that rely on padding to make a plaintext message a perfect multiple of the cipher's block length, like CBC Mode, can be susceptible to divulging the entire contents of a message through a padding oracle. This can be an *extremely* serious attack.
Password	A string of characters (letters, numbers and other symbols) that are used to authenticate an identity or to verify access authorization. A passphrase is a special case of a password that is a sequence of words or other text. In this document, the use of the term "password' includes this special case.
Permutation	An ordered (re)arrangement of the elements of a set.
PKCS	PKCS Public Key Cryptography Standard.
PKI	Public-Key Infrastructure
PSS	Probabilistic Signature Scheme
Plaintext data	Intelligible data that has meaning and can be understood without the application of decryption.
Preimage resistance	Given a randomly chosen hash value, *hash_value*, it is computationally infeasible to find an *x* so that *hash(x)* = *hash_value*. This property is also called the one-way property. Preimage resistance is measured by the

	amount of work that would be needed to find a preimage for a cryptographic hash function with high probability. If the amount of work is 2^N, then the preimage resistance is N bits. The estimated strength for preimage resistance provided by a hash-function is the length of the hash value produced by a given cryptographic hash function, i.e., the estimated security strength for preimage resistance is L bits. For example, SHA-256 produces a (full-length) hash value of 256 bits; SHA-256 provides an estimated preimage resistance of 256 bits.
Pre-shared key	A key that is already known by the entities needing to use it.
Privacy	Assurance that the confidentiality of, and access to, certain information about an entity is protected.
Private key	A cryptographic key, used with a public key cryptographic algorithm that is uniquely associated with an entity and is not made public. In an asymmetric-key (public key) cryptosystem, the private key is associated with a public key. Depending on the algorithm, the private key may be used to: 1. Compute the corresponding public key, 2. Compute a digital signature that may be verified using the corresponding public key, 3. Decrypt data that was encrypted using the corresponding public key, or 4. Compute a key-derivation key, which may then be used as an input to a key derivation process.
Public-key certificate	A set of data that uniquely identifies an entity, contains the entity's public key and possibly other information, and is digitally signed by a trusted party, thereby binding the public key to the entity. Additional information in the certificate could specify how the key is used and its cryptoperiod.
Pseudorandom function	A function that can be used to generate output from a secret random seed and a data variable, such that the output is computationally indistinguishable from truly random output.
Public key	A cryptographic key used with a public key cryptographic algorithm that is uniquely associated with an entity and that may be made public. In an

	asymmetric key (public key) cryptosystem, the public key is associated with a private key. The public key may be known by anyone and, depending on the algorithm, may be used to: 1. Verify a digital signature that is signed by the corresponding private key, 2. Encrypt data that can be decrypted by the corresponding private key, or 3. Compute a piece of shared data (i.e., data that is known only by two or more specific entities).
Public key algorithm	See asymmetric key algorithm.
RSA	Rivest-Shamir-Adelman. An asymmetric algorithm.
RFC	Request For Comment
Random Bit Generator (RBG)	A device or algorithm that outputs bits that appear to be "statistically independent" and unbiased.
Random number generator (RNG)	A process used to generate an unpredictable series of numbers. Also, referred to as a Random bit generator (RBG).
Replay Attack	A generalized networking attack that retransmits an old message to a receiver in an attempt to trick the receiver into believing the message is valid (or was just transmitted in real time).
RNG seed	A seed that is used to initialize a deterministic random bit generator.
Rootkit	Malware that enables unauthorized, privileged access to a computer while actively hiding its presence from administrators by subverting standard operating-system functionality or other applications.
SHA-256	SHA-256 Secure Hash Algorithm with a 256-bit output.
SHA-512/X	SHA-512 Secure Hash Algorithm with X bits of output.
SSH	Secure Shell

S-BOX	A function that transforms a fixed number of input bits into a (possibly different) fixed number of output bits.
Scalability	The ability of a system to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.
Second Preimage Resistance	It is computationally infeasible to find a second input that has the same hash value as any other specified input. That is, given an input *x*, it is computationally infeasible to find a second input *x′* that is different from *x*, such that *hash*(*x*) = *hash* (*x′*). Second preimage resistance is measured by the amount of work that would be needed to find a second preimage for a cryptographic hash function with high probability.
Secret key	A single cryptographic key that is used with a secret key (symmetric key) cryptographic algorithm that is uniquely associated with one or more entities and is not made public.
Security Domain	A group of entities that have common goals and requirements (including security considerations) that have been specified in a common security policy.
Security strength	A number associated with the amount of work (that is, the number of basic operations of some sort) that is required to break a cryptographic algorithm or system. A security strength is often expressed in bits. If the security strength is *S* bits, then it is expected that (roughly) 2*S* basic operations are required to break it.
Seed	A secret value that is used to initialize a process (e.g., a deterministic random bit generator). Also see RNG seed.
Self-signed certificate	A public-key certificate whose digital signature may be verified by the public key contained within the certificate. The signature on a self-signed certificate protects the integrity of the data, but does not guarantee the authenticity of the information. The trust of self-signed certificates is based on the secure procedures used to distribute them.
Shared secret	A secret value that has been computed using a key establishment scheme and is used as input to a key

	derivation function or extraction-then-expansion procedure.
Side-Channel Attack	An attack that seeks to exploit peripheral aspects of a system or an algorithm that expose useful information to the attacker.
Signature generation	The use of a digital signature algorithm and a private key to generate a digital signature on data.
Signature verification	The use of a digital signature algorithm and a public key to verify a digital signature on data.
Split knowledge	A process by which a cryptographic key is split into *n* multiple key components, individually providing no knowledge of the original key, which can be subsequently combined to recreate the original cryptographic key. If knowledge of *k* (where *k* is less than or equal to *n*) components is required to construct the original key, then knowledge of any *k*-1 key components provides no information about the original key other than, possibly, its length. Note that in this document, split knowledge is not intended to cover key shares, such as those used in threshold or multi-party signatures.
Static key	A key that is intended for use for a relatively long period of time and is typically intended for use in many instances of a cryptographic key establishment scheme. Contrast with an ephemeral key.
Symmetric key	A single cryptographic key that is used with a symmetric key (secret key) algorithm that is uniquely associated with one or more entities and is not made public.
Symmetric key algorithm	A cryptographic algorithm that uses the same secret key for its operation and, if applicable, for reversing the effects of the operation (e.g., an HMAC key for keyed hashing, or an AES key for encryption and decryption).
TDEA	Triple Data Encryption Standard
TLS	Transport Layer Security

Timing Attack	An attack that uses the amount of time a particular cryptographic operation takes to execute.
Traffic Analysis	A generic term for studying a communication channel to learn which parties are communicating, with whom they are communicating, and the frequency and size of the messages
Trust	A characteristic of an entity that indicates its ability to perform certain functions or services correctly, fairly, and impartially, along with assurance that the entity and its identifier are genuine.
Trusted Channel	A trusted and safe communication channel used to share sensitive information between two entities that are not collocated in a secure facility.
Trusted Party	A party that is trusted by its clients to generate cryptographic keys.
Unobservability	Assurance that an observer is unable to identify or make inferences about the parties involved in a transaction.
User	An individual authorized by an organization and its policies to use an information system, one or more of its applications, and its security procedures and services. Also see Entity.
Verification Oracle	An oracle allowing the attacker to attempt to verify some type of data such as a hash or a MAC. The oracle lets the attacker know whether the verification process succeeded or failed, and in some cases, the time it took to fail or at which point in the process the failure occurred. Where a verification oracle can accurately gauge the underlying function's timing, this can open the door to a *timing attack*.
Work	The expected time to break a cipher with a given resource. For example, 12 MIPS years would be the amount of work that one computer, with the capability of processing a million instructions per second, could do in 12 years. The same amount of work could be done by 12 such computers in one year, assuming that the algorithm being executed can be sufficiently parallelized.

X.509 certificate	The X.509 public-key certificate or the X.509 attribute certificate, as defined by the ISO/ITU-T X.509 standard. Most commonly (including in this document), an X.509 certificate refers to the X.509 public-key certificate.
XML	Extensible Markup Language

Index

About the Author

Logan Gore is the Senior Solutions Architect at Gretl, Inc., a Seattle-based tech company. He has written several projects and white papers on cryptography and secure development. Logan holds certifications in enterprise security, cloud technologies, data storage networking, server and infrastructure, as well as developer certifications for C#/.NET and C++.

linkedin.com/in/gorelogan

github.com/logangore