Chapter Review

Central to securing our systems is understanding their components and how they interact with each other—in other words, their architectures. While it may seem that architectural terminology overlaps quite a bit, in reality each approach brings some unique challenges and some not-so-unique challenges. As security professionals, we need to understand where architectures are similar and where they differ. We can mix and match, of course, but must also do so with a clear understanding of the underlying issues. In this chapter, we've classified the more common system architectures and discussed what makes them unique and what specific security challenges they pose. Odds are that you will encounter devices and systems in most, if not all, of the architectures we've covered here.

Quick Review

- Client-based systems execute all their core functions on the user's device and don't require network connectivity.
- Server-based systems require that a client make requests from a server across a network connection.
- Transactions are sequences of actions required to properly change the state of a database.
- Database transactions must be atomic, consistent, isolated, and durable (ACID).
- Aggregation is the act of combining information from separate sources and is a security problem when it allows unauthorized individuals to piece together sensitive information.
- Inference is deducing a whole set of information from a subset of its aggregated components. This is a security problem when it allows unauthorized individuals to infer sensitive information.
- High-performance computing (HPC) is the aggregation of computing power in ways that exceed the capabilities of general-purpose computers for the specific purpose of solving large problems.
- Industrial control systems (ICS) consist of information technology that is specifically designed to control physical devices in industrial processes.
- Any system in which computers and physical devices collaborate via the exchange of inputs and outputs to accomplish a task or objective is an embedded or cyber-physical system.
- The two main types of ICS are distributed control systems (DCS) and supervisory control and data acquisition (SCADA) systems. The main difference between them is that a DCS controls local processes while SCADA is used to control things remotely.
- ICS should always be logically or physically isolated from public networks.
- Virtualized systems are those that exist in software-simulated environments.
- Virtual machines (VMs) are systems in which the computing hardware has been virtualized for the operating systems running in them.

- Containers are systems in which the operating systems have been virtualized for the applications running in them.
- Microservices are software architectures in which features are divided into multiple separate components that work together in a distributed manner across a network.
- Containers and microservices don't have to be used together but it's very common to do so.
- In a serverless architecture, the services offered to end users can be performed without a requirement to set up any dedicated server infrastructure.
- Cloud computing is the use of shared, remote computing devices for the purpose of providing improved efficiencies, performance, reliability, scalability, and security.
- Software as a Service (SaaS) is a cloud computing model that provides users access to a specific application that executes in the service provider's environment.
- Platform as a Service (PaaS) is a cloud computing model that provides users access to a computing platform but not to the operating system or to the virtual machine on which it runs.
- Infrastructure as a Service (IaaS) is a cloud computing model that provides users unfettered access to a cloud device, such as an instance of a server, which includes both the operating system and the virtual machine on which it runs.
- An embedded system is a self-contained, typically ruggedized, computer system with its own processor, memory, and input/output devices that is designed for a very specific purpose.
- The Internet of Things (IoT) is the global network of connected embedded systems.
- A distributed system is a system in which multiple computing nodes, interconnected by a network, exchange information for the accomplishment of collective tasks.
- Edge computing is a distributed system in which some computational and data storage assets are deployed close to where they are needed in order to reduce latency and network traffic.

Questions

Please remember that these questions are formatted and asked in a certain way for a reason. Keep in mind that the CISSP exam is asking questions at a conceptual level. Questions may not always have the perfect answer, and the candidate is advised against always looking for the perfect answer. Instead, the candidate should look for the best answer in the list.

1. Which of the following lists two foundational properties of database transactions?

A. Aggregation and inference

B. Scalability and durability

C. Consistency and performance

D. Atomicity and isolation

2. Which of the following is *not* true about containers?
   - **A.** They are embedded systems.
   - **B.** They are virtualized systems.
   - **C.** They commonly house microservices.
   - **D.** They operate in a sandbox.
3. What is the term that describes a database attack in which an unauthorized user is able to combine information from separate sources to learn sensitive information to which the user should not have access?
   - **A.** Aggregation
   - **B.** Containerization
   - **C.** Serialization
   - **D.** Collection
4. What is the main difference between a distributed control system (DCS) and supervisory control and data acquisition (SCADA)?
   - **A.** SCADA is a type of industrial control system (ICS), while a DCS is a type of bus.
   - **B.** SCADA controls systems in close proximity, while a DCS controls physically distant ones.
   - **C.** A DCS controls systems in close proximity, while SCADA controls physically distant ones.
   - **D.** A DCS uses programmable logic controllers (PLCs), while SCADA uses remote terminal units (RTUs).
5. What is the main purpose of a hypervisor?
   - **A.** Virtualize hardware resources and manage virtual machines
   - **B.** Virtualize the operating system and manage containers
   - **C.** Provide visibility into virtual machines for access control and logging
   - **D.** Provide visibility into containers for access control and logging
6. Which cloud service model provides customers direct access to hardware, the network, and storage?
   - **A.** SaaS
   - **B.** PaaS
   - **C.** IaaS
   - **D.** FaaS
7. Which cloud service model do you recommend to enable access to developers to write custom code while also providing all employees access from remote offices?
   - **A.** PaaS
   - **B.** SaaS

C. FaaS

D. IaaS

8. Which of the following is not a major issue when securing embedded systems?

A. Use of proprietary code

B. Devices that "phone home"

C. Lack of microcontrollers

D. Ability to update and patch them securely

9. Which of the following is true about edge computing?

A. Uses no centralized computing resources, pushing all computation to the edge

B. Pushes computation to the edge while retaining centralized data management

C. Typically consists of two layers: end devices and cloud infrastructure

D. Is an evolution of content distribution networks

Use the following scenario to answer Questions 10–12. You were just hired as director of cybersecurity for an electric power company with facilities around your country. Carmen is the director of operations and offers to give you a tour so you can see the security measures that are in place on the operational technology (OT).

10. What system would be used to control power generation, distribution, and delivery to all your customers?

A. Supervisory control and data acquisition (SCADA)

B. Distributed control system (DCS)

C. Programmable logic controller

D. Edge computing system

11. You see a new engineer being coached remotely by a more senior member of the staff in the use of the human-machine interface (HMI). Carmen tells you that senior engineers are allowed to access the HMI from their personal computers at home to facilitate this sort of impromptu training. She asks what you think of this policy. How should you respond?

A. Change the policy. They should not access the HMI with their personal computers, but they could do so using a company laptop, assuming they also use a virtual private network (VPN).

B. Change the policy. ICS devices should always be isolated from the Internet.

C. It is acceptable because the HMI is only used for administrative purposes and not operational functions.

D. It is acceptable because safety is the fundamental concern in ICS, so it is best to let the senior engineers be available to train other staff from home.

12. You notice that several ICS devices have never been patched. When you ask why, Carmen tells you that those are mission-critical devices, and her team has no way of testing the patches before patching these production systems. Fearing that patching them could cause unexpected outages or, worse, injure someone, she has authorized them to remain as they are. Carmen asks whether you agree. How could you respond?
    - **A.** Yes. As long as we document the risk and ensure the devices are as isolated and as closely monitored as possible.
    - **B.** Yes. Safety and availability trump all other concerns when it comes to ICS security.
    - **C.** No. You should stand up a testing environment so you can safely test the patches and then deploy them to all devices.
    - **D.** No. These are critical devices and should be patched as soon as possible.

Answers

1. **D.** The foundational properties of database transactions are atomicity, consistency, isolation, and durability (ACID).
2. **A.** Containers are virtualized systems that commonly (though not always) house microservices and run in sandboxes. It would be highly unusual to implement a container as an embedded system.
3. **A.** Aggregation happens when a user does not have the clearance or permission to access specific information, but she does have the permission to access components of this information. She can then figure out the rest and obtain restricted information.
4. **C.** The main difference is that a DCS controls devices within fairly close proximity, while SCADA controls large-scale physical processes involving nodes separated by significant distances. They both can (and frequently use) PLCs, but RTUs are almost always seen in SCADA systems.
5. **A.** Hypervisors are almost always used to virtualize the hardware on which virtual machines run. They can also provide visibility and logging, but these are secondary functions. Containers are the equivalents of hypervisors, but they work at a higher level by virtualizing the operating system.
6. **C.** Infrastructure as a Service (IaaS) offers an effective and affordable way for organizations to get all the benefits of managing their own hardware without the massive overhead costs associated with acquisition, physical storage, and disposal of the hardware.
7. **A.** Platform as a Service (PaaS) solutions are optimized to provide value focused on software development, offering direct access to a development environment to enable an organization to build its own solutions on the cloud infrastructure, rather than providing its own infrastructure.

8. **C.** Embedded systems are usually built around microcontrollers, which are specialized devices that consist of a CPU, memory, and peripheral control interfaces. All the other answers are major issues in securing embedded systems.
9. **D.** Edge computing is an evolution of content distribution networks, which were designed to bring web content closer to its clients. It is a distributed system in which some computational and data storage assets are deployed close to where they are needed in order to reduce latency and network traffic. Accordingly, some computing and data management is handled in each of three different layers: end devices, edge devices, and cloud infrastructure.
10. **A.** SCADA was designed to control large-scale physical processes involving nodes separated by significant distances, as is the case with electric power providers.
11. **B.** It is a best practice to completely isolate ICS devices from Internet access. Sometimes this is not possible for operational reasons, so remote access through a VPN could be allowed even though it is not ideal.
12. **A.** It is all too often the case that organizations can afford neither the risk of pushing untested patches to ICS devices nor the costs of standing up a testing environment. In these conditions, the best strategy is to isolate and monitor the devices as much as possible.

This page intentionally left blank

CHAPTER 8

Cryptology

This chapter presents the following:

- Principles of cryptology
- Symmetric cryptography
- Asymmetric cryptography
- Public key infrastructure
- Cryptanalytic attacks

Three can keep a secret, if two of them are dead.

—Benjamin Franklin

Now that you have a pretty good understanding of system architectures from Chapter 7, we turn to a topic that is central to protecting these architectures. *Cryptography* is the practice of storing and transmitting information in a form that only authorized parties can understand. Properly designed and implemented, cryptography is an effective way to protect sensitive data throughout its life cycle. However, with enough time, resources, and motivation, hackers can successfully attack most cryptosystems and reveal the information. So, a more realistic goal of cryptography is to make obtaining the information too work intensive or time consuming to be worthwhile to the attacker.

Cryptanalysis is the name collectively given to techniques that aim to weaken or defeat cryptography. This is what the adversary attempts to do to thwart the defender's use of cryptography. Together, cryptography and cryptanalysis comprise *cryptology*. In this chapter, we'll take a good look at both sides of this topic. This is an important chapter in the book, because we can't defend our information systems effectively without understanding applied cryptology.

The History of Cryptography

Cryptography has roots in antiquity. Around 600 B.C., Hebrews invented a cryptographic method called *atbash* that required the alphabet to be flipped so each letter in the original message was mapped to a different letter in the flipped, or shifted, message. An example of an encryption key used in the atbash encryption scheme is shown here:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

If you want to encrypt the word "security" you would instead use "hvxfirgb." Atbash is an example of a *substitution cipher* because each character is replaced with another character. This type of substitution cipher is referred to as a *monoalphabetic substitution cipher* because it uses only one alphabet, whereas a *polyalphabetic substitution cipher* uses multiple alphabets.

TIP Cipher is another term for algorithm.

Around 400 B.C., the Spartans used a system of encrypting information in which they would write a message on a sheet of papyrus (a type of paper) that was wrapped around a staff (a stick or wooden rod), which was then delivered and wrapped around a different staff by the recipient. The message was only readable if it was wrapped around the correct size staff, which made the letters properly match up, as shown in Figure 8-1. When the papyrus was not wrapped around the staff, the writing appeared as just a bunch of random characters. This approach, known as the *scytale cipher*, is an example of a *transposition cipher* because it relies on changing the sequence of the characters to obscure their meaning. Only someone who knows how to rearrange them would be able to recover the original message.

Later, in Rome, Julius Caesar (100–44 B.C.) developed a simple method of shifting letters of the alphabet, similar to the atbash scheme. He simply shifted the alphabet by three positions. The following example shows a standard alphabet and a shifted alphabet. The alphabet serves as the algorithm, and the key is the number of locations it has been shifted during the encryption and decryption process.

- **Standard alphabet:**
  `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
- **Cryptographic alphabet:**
  `DEFGHIJKLMNOPQRSTUVWXYZABC`

As an example, suppose we need to encrypt the message "MISSION ACCOMPLISHED." We take the first letter of this message, *M*, and shift up three locations within the alphabet. The encrypted version of this first letter is *P*, so we write

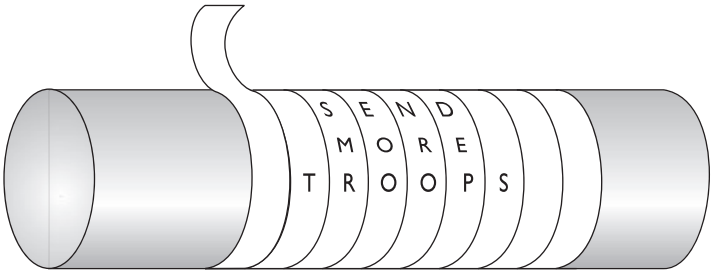


Figure 8-1 The scytale was used by the Spartans to decipher encrypted messages.

that down. The next letter to be encrypted is *I*, which matches *L* when we shift three spaces. We continue this process for the whole message. Once the message is encrypted, a carrier takes the encrypted version to the destination, where the process is reversed.

- **Original message:**
  `MISSION ACCOMPLISHED`
- **Encrypted message:**
  `PLVVLRQ DFFRPSOLVKHG`

Today, this technique seems too simplistic to be effective, but in the time of Julius Caesar, not very many people could read in the first place, so it provided a high level of protection. The Caesar cipher, like the atbash cipher, is an example of a monoalphabetic cipher. Once more people could read and reverse-engineer this type of encryption process, the cryptographers of that day increased the complexity by creating polyalphabetic ciphers.

In the 16th century in France, Blaise de Vigenère developed a polyalphabetic substitution cipher for Henry III. This was based on the Caesar cipher, but it increased the difficulty of the encryption and decryption process. As shown in Figure 8-2, we have a message that needs to be encrypted, which is SYSTEM SECURITY AND CONTROL. We have a key with the value of SECURITY. We also have a Vigenère table, or algorithm, which is really the Caesar cipher on steroids. Whereas the Caesar cipher used a single shift alphabet (letters were shifted up three places), the Vigenère cipher has 27 shift alphabets and the letters are shifted up only one place.

So, looking at the example in Figure 8-2, we take the first value of the key, *S*, and starting with the first alphabet in our algorithm, trace over to the *S* column. Then we look at the first character of the original message that needs to be encrypted, which is *S*, and go down to the *S* row. We follow the column and row and see that they intersect on the value *K*. That is the first encrypted value of our message, so we write down *K*. Then we go to the next value in our key, which is *E*, and the next character in the original message, which is *Y*. We see that the *E* column and the *Y* row intersect at the cell with the value of *C*. This is our second encrypted value, so we write that down. We continue this process for the whole message (notice that the key repeats itself, since the message is longer than the key). The result is an encrypted message that is sent to the destination. The destination must have the same algorithm (Vigenère table) and the same key (SECURITY) to properly reverse the process to obtain a meaningful message.

The evolution of cryptography continued as countries refined it using new methods, tools, and practices with varying degrees of success. Mary, Queen of Scots, lost her life in the 16th century when an encrypted message she sent was intercepted. During the American Revolutionary War, Benedict Arnold used a codebook cipher to exchange information on troop movement and strategic military advancements. By the late 1800s, cryptography was commonly used in the methods of communication between military factions.

During World War II, encryption devices were used for tactical communication, which drastically improved with the mechanical and electromechanical technology that provided the world with telegraphic and radio communication. The rotor cipher machine, which is a device that substitutes letters using different rotors within the

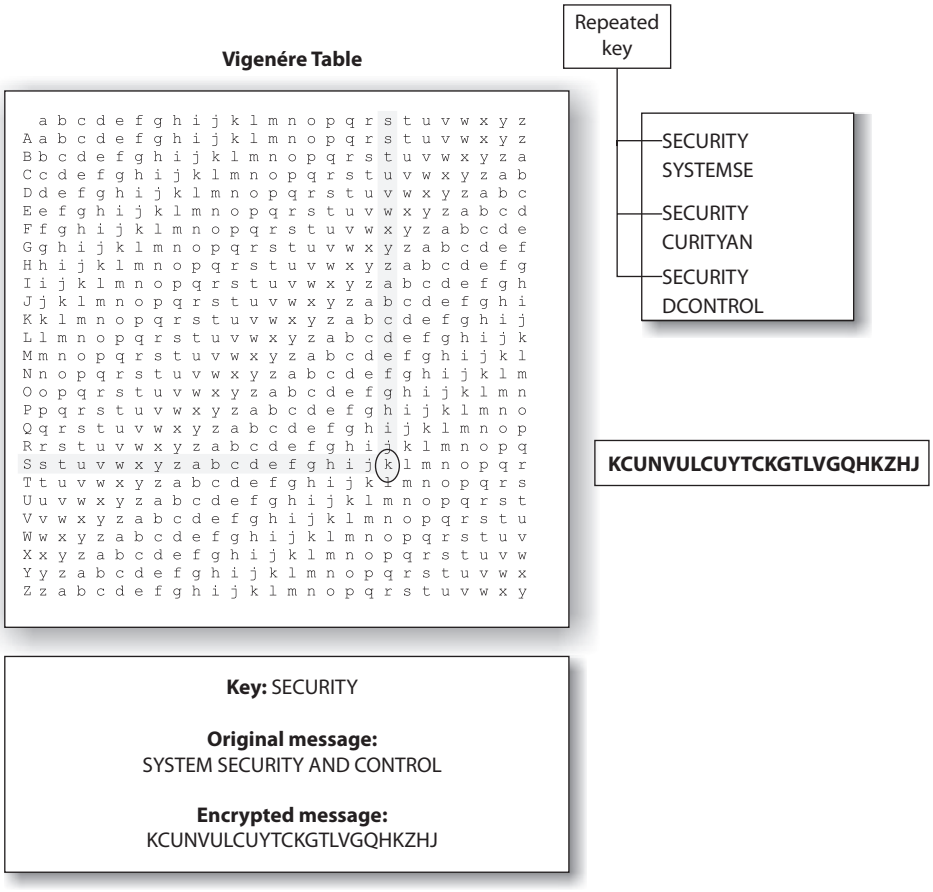


Figure 8-2 Polyalphabetic algorithms were developed to increase encryption complexity.

machine, was a huge breakthrough in military cryptography that provided complexity that proved difficult to break. This work gave way to the most famous cipher machine in history to date: Germany's *Enigma* machine. The Enigma machine had separate rotors, a plug board, and a reflecting rotor.

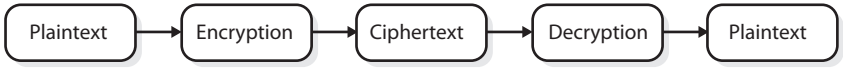
The originator of the message would configure the Enigma machine to its initial settings before starting the encryption process. The operator would type in the first letter of the message, and the machine would substitute the letter with a different letter and present it to the operator. This encryption was done by moving the rotors a predefined number of times. So, if the operator typed in a *T* as the first character, the Enigma machine might present an *M* as the substitution value. The operator would write down the letter *M* on his sheet. The operator would then advance the rotors and enter the next letter. Each time a new letter was to be encrypted, the operator would advance the rotors to a new setting. This process was followed until the whole message was encrypted. Then the encrypted text was transmitted over the airwaves, most likely to a German U-boat. The chosen

substitution for each letter was dependent upon the rotor setting, so the crucial and secret part of this process (the key) was the initial setting and how the operators advanced the rotors when encrypting and decrypting a message. The operators at each end needed to know this sequence of increments to advance each rotor in order to enable the German military units to properly communicate.

When computers were invented, the possibilities for encryption methods and devices expanded exponentially and cryptography efforts increased dramatically. This era brought unprecedented opportunity for cryptographic designers to develop new encryption techniques. A well-known and successful project was *Lucifer*, which was developed at IBM. Lucifer introduced complex mathematical equations and functions that were later adopted and modified by the U.S. National Security Agency (NSA) to establish the U.S. Data Encryption Standard (DES) in 1976, a federal government standard. DES was used worldwide for financial and other transactions, and was embedded into numerous commercial applications. Though it was cracked in the late 1990s and is no longer considered secure, DES represented a significant advancement for cryptography. It was replaced a few years later by the Advanced Encryption Standard (AES), which continues to protect sensitive data to this day.

Cryptography Definitions and Concepts

Encryption is a method of transforming readable data, called *plaintext*, into a form that appears to be random and unreadable, which is called *ciphertext*. Plaintext is in a form that can be understood either by a person (a document) or by a computer (executable code). Once plaintext is transformed into ciphertext, neither human nor machine can properly process it until it is decrypted. This enables the transmission of confidential information over insecure channels without unauthorized disclosure. When sensitive data is stored on a computer, it is usually protected by logical and physical access controls. When this same sensitive information is sent over a network, it no longer has the advantage of these controls and is in a much more vulnerable state.



A system or product that provides encryption and decryption is referred to as a *cryptosystem* and can be created through hardware components or program code in an application. The cryptosystem uses an encryption algorithm (which determines how simple or complex the encryption process will be), keys, and the necessary software components and protocols. Most algorithms are complex mathematical formulas that are applied in a specific sequence to the plaintext. Most encryption methods use a secret value called a key (usually a long string of bits), which works with the algorithm to encrypt and decrypt the text.

The *algorithm*, the set of rules also known as the *cipher*, dictates how enciphering and deciphering take place. Many of the mathematical algorithms used in computer systems today are publicly known and are not the secret part of the encryption process. If the internal mechanisms of the algorithm are not a secret, then something must be: the key.

A common analogy used to illustrate this point is the use of locks you would purchase from your local hardware store. Let's say 20 people bought the same brand of lock. Just because these people share the same type and brand of lock does not mean they can now unlock each other's doors and gain access to their private possessions. Instead, each lock comes with its own key, and that one key can open only that one specific lock.

In encryption, the *key* (also known as *cryptovariable*) is a value that comprises a large sequence of random bits. Is it just any random number of bits crammed together? Not really. An algorithm contains a *keyspace*, which is a range of values that can be used to construct a key. When the algorithm needs to generate a new key, it uses random values from this keyspace. The larger the keyspace, the more available values that can be used to represent different keys—and the more random the keys are, the harder it is for intruders to figure them out. For example, if an algorithm allows a key length of 2 bits, the keyspace for that algorithm would be 4, which indicates the total number of different keys that would be possible. (Remember that we are working in binary and that 2^2 equals 4.) That would not be a very large keyspace, and certainly it would not take an attacker very long to find the correct key that was used.

A large keyspace allows for more possible keys. (Today, we are commonly using key sizes of 128, 256, 512, or even 1,024 bits and larger.) So a key size of 512 bits would provide 2^{512} possible combinations (the keyspace). The encryption algorithm should use the entire keyspace and choose the values to make up the keys as randomly as possible. If a smaller keyspace were used, there would be fewer values to choose from when generating a key, as shown in Figure 8-3. This would increase an attacker's chances of figuring out the key value and deciphering the protected information.

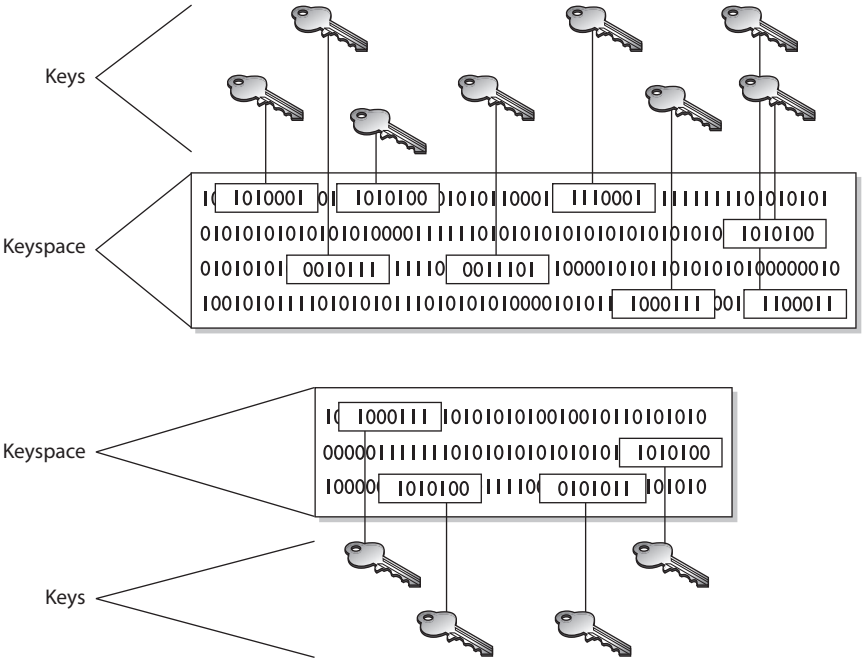


Figure 8-3 Larger keyspaces permit a greater number of possible key values.

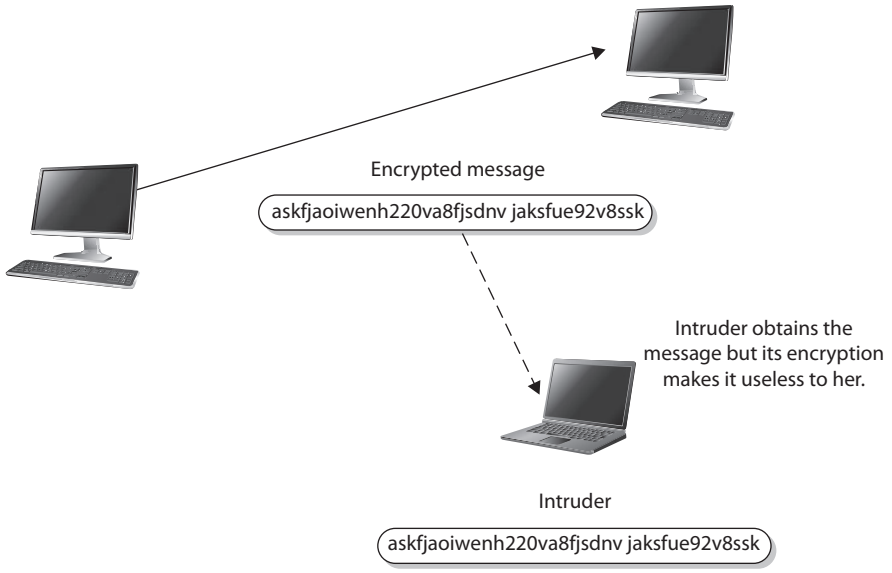


Figure 8-4 Without the right key, the captured message is useless to an attacker.

If an eavesdropper captures a message as it passes between two people, she can view the message, but it appears in its encrypted form and is therefore unusable. Even if this attacker knows the algorithm that the two people are using to encrypt and decrypt their information, without the key, this information remains useless to the eavesdropper, as shown in Figure 8-4.

Cryptosystems

A *cryptosystem* encompasses all of the necessary components for encryption and decryption to take place. Pretty Good Privacy (PGP) is just one example of a cryptosystem. A cryptosystem is made up of at least the following:

- Software
- Protocols
- Algorithms
- Keys

Cryptosystems can provide the following services:

- **Confidentiality** Renders the information unintelligible except by authorized entities.
- **Integrity** Ensures that data has not been altered in an unauthorized manner since it was created, transmitted, or stored.
- **Authentication** Verifies the identity of the user or system that created the information.

- **Authorization** Provides access to some resource to the authenticated user or system.
- **Nonrepudiation** Ensures that the sender cannot deny sending the message.

As an example of how these services work, suppose your boss sends you an e-mail message stating that you will be receiving a raise that doubles your salary. The message is encrypted, so you can be sure it really came from your boss (authenticity), that someone did not alter it before it arrived at your computer (integrity), that no one else was able to read it as it traveled over the network (confidentiality), and that your boss cannot deny sending it later when he comes to his senses (nonrepudiation).

Different types of messages and transactions require higher or lower degrees of one or all of the services that cryptography methods can supply. Military and intelligence agencies are very concerned about keeping information confidential, so they would choose encryption mechanisms that provide a high degree of secrecy. Financial institutions care about confidentiality, but they also care about the integrity of the data being transmitted, so the encryption mechanism they would choose may differ from the military's encryption methods. If messages were accepted that had a misplaced decimal point or zero, the ramifications could be far reaching in the financial world. Legal agencies may care most about the authenticity of the messages they receive. If information received ever needed to be presented in a court of law, its authenticity would certainly be questioned; therefore, the encryption method used must ensure authenticity, which confirms who sent the information.

NOTE If David sends a message and then later claims he did not send it, this is an act of repudiation. When a cryptography mechanism provides nonrepudiation, the sender cannot later deny he sent the message (well, he can try to deny it, but the cryptosystem proves otherwise).

The types and uses of cryptography have increased over the years. At one time, cryptography was mainly used to keep secrets secret (confidentiality), but today we use cryptography to ensure the integrity of data, to authenticate messages, to confirm that a message was received, to provide access control, and much more.

Kerckhoffs' Principle

Auguste Kerckhoffs published a paper in 1883 stating that the only secrecy involved with a cryptography system should be the key. He claimed that the algorithm should be publicly known. He asserted that if security were based on too many secrets, there would be more vulnerabilities to possibly exploit.

So, why do we care what some guy said almost 140 years ago? Because this debate is still going on. Cryptographers in certain sectors agree with *Kerckhoffs' principle*, because making an algorithm publicly available means that many more people can view the source code, test it, and uncover any type of flaws or weaknesses. It is the attitude of "many heads are better than one." Once someone uncovers some type of flaw, the developer can fix the issue and provide society with a much stronger algorithm.

But not everyone agrees with this philosophy. Governments around the world create their own algorithms that are not released to the public. Their stance is that if a smaller number of people know how the algorithm actually works, then a smaller number of people will know how to possibly break it. Cryptographers in the private sector do not agree with this practice and do not commonly trust algorithms they cannot examine. It is basically the same as the open-source versus compiled software debate that is in full force today.

The Strength of the Cryptosystem

The *strength* of an encryption method comes from the algorithm, the secrecy of the key, the length of the key, and how they all work together within the cryptosystem. When strength is discussed in encryption, it refers to how hard it is to figure out the algorithm or key, whichever is not made public. Attempts to break a cryptosystem usually involve processing an amazing number of possible values in the hopes of finding the one value (key) that can be used to decrypt a specific message. The strength of an encryption method correlates to the amount of necessary processing power, resources, and time required to break the cryptosystem or to figure out the value of the key.

Breaking a cryptosystem can be accomplished by a *brute-force attack*, which means trying every possible key value until the resulting plaintext is meaningful. Depending on the algorithm and length of the key, this can be an easy task or one that is close to impossible. If a key can be broken with an Intel Core i5 processor in three hours, the cipher is not strong at all. If the key can only be broken with the use of a thousand multiprocessing systems over 1.2 million years, then it is pretty darned strong. The introduction of commodity cloud computing has really increased the threat of brute-force attacks.

The goal when designing an encryption method is to make compromising it too expensive or too time consuming. Another name for cryptography strength is *work factor*, which is an estimate of the effort and resources it would take an attacker to penetrate a cryptosystem.

Even if the algorithm is very complex and thorough, other issues within encryption can weaken encryption methods. Because the key is usually the secret value needed to actually encrypt and decrypt messages, improper protection of the key can weaken the encryption. Even if a user employs an algorithm that has all the requirements for strong encryption, including a large keyspace and a large and random key value, if she shares her key with others, the strength of the algorithm becomes almost irrelevant.

Important elements of encryption are to use an algorithm without flaws, use a large key size, use all possible values within the keyspace selected as randomly as possible, and protect the actual key. If one element is weak, it could be the link that dooms the whole process.

One-Time Pad

A *one-time pad* is a perfect encryption scheme because it is considered unbreakable if implemented properly. It was invented by Gilbert Vernam in 1917, so sometimes it is referred to as the Vernam cipher.

This cipher does not use shift alphabets, as do the Caesar and Vigenère ciphers discussed earlier, but instead uses a pad made up of random values, as shown in Figure 8-5. Our plaintext message that needs to be encrypted has been converted into bits, and our one-time

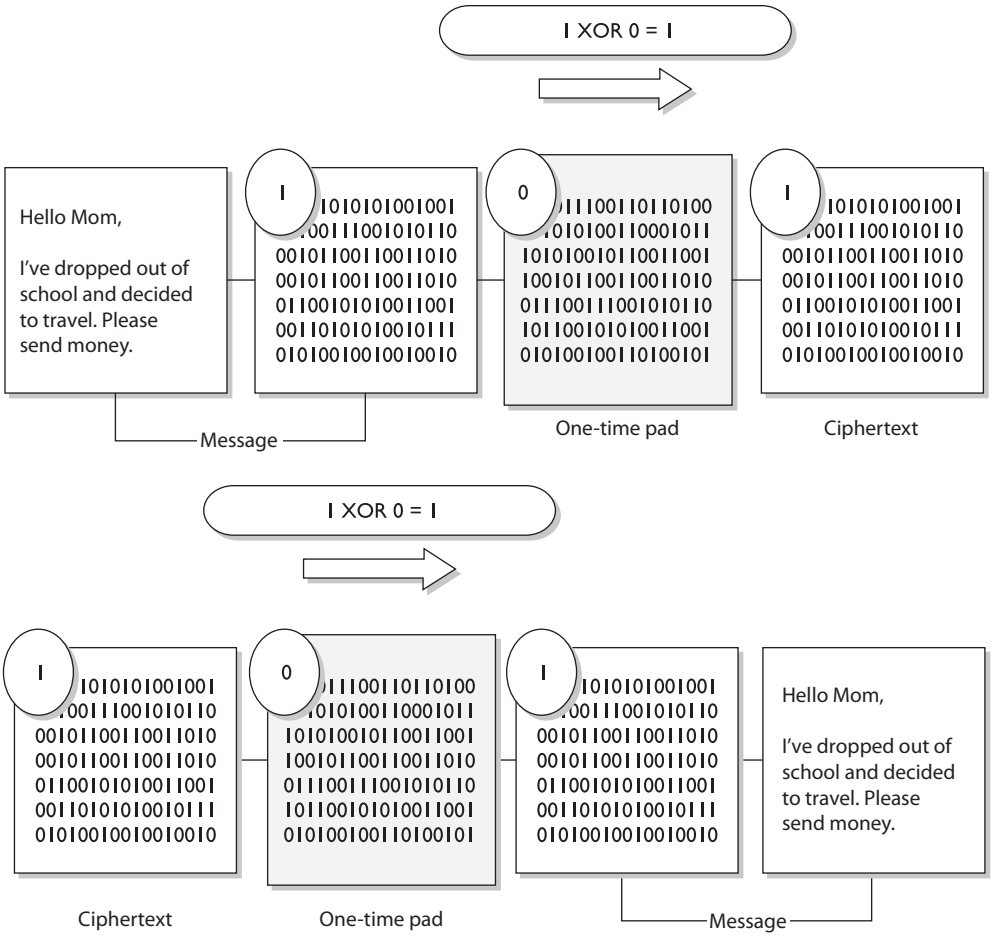


Figure 8-5 A one-time pad

pad is made up of random bits. This encryption process uses a binary mathematic function called exclusive-OR, usually abbreviated as XOR.

XOR is an operation that is applied to 2 bits and is a function commonly used in binary mathematics and encryption methods. When combining the bits, if both values are the same, the result is 0 (1 XOR 1 = 0). If the bits are different from each other, the result is 1 (1 XOR 0 = 1). For example:

Message stream:	1	0	0	1	0	1	0	1	1	1
Keystream:	0	0	1	1	1	0	1	0	1	0
Ciphertext stream:	1	0	1	0	1	1	1	1	0	1

So in our example, the first bit of the message is XORed to the first bit of the one-time pad, which results in the ciphertext value 1. The second bit of the message is XORed with

the second bit of the pad, which results in the value 0. This process continues until the whole message is encrypted. The result is the encrypted message that is sent to the receiver.

In Figure 8-5, we also see that the receiver must have the same one-time pad to decrypt the message by reversing the process. The receiver takes the first bit of the encrypted message and XORs it with the first bit of the pad. This results in the plaintext value. The receiver continues this process for the whole encrypted message until the entire message is decrypted.

The one-time pad encryption scheme is deemed unbreakable only if the following things are true about the implementation process:

- *The pad must be used only one time.* If the pad is used more than one time, this might introduce patterns in the encryption process that will aid the eavesdropper in his goal of breaking the encryption.
- *The pad must be at least as long as the message.* If it is not as long as the message, the pad will need to be reused to cover the whole message. This would be the same thing as using a pad more than one time, which could introduce patterns.
- *The pad must be securely distributed and protected at its destination.* This is a very cumbersome process to accomplish, because the pads are usually just individual pieces of paper that need to be delivered by a secure courier and properly guarded at each destination.
- *The pad must be made up of truly random values.* This may not seem like a difficult task, but even our computer systems today do not have truly random number generators; rather, they have pseudorandom number generators.

NOTE Generating truly random numbers is very difficult. Most systems use an algorithmic *pseudorandom number generator (PRNG)* that takes as its input a seed value and creates a stream of pseudorandom values from it. Given the same seed, a PRNG generates the same sequence of values. Truly random numbers must be based on natural phenomena such as thermal noise and quantum mechanics.

Although the one-time pad approach to encryption can provide a very high degree of security, it is impractical in most situations because of all of its different requirements. Each possible pair of entities that might want to communicate in this fashion must receive, in a secure fashion, a pad that is as long as, or longer than, the actual message. This type of key management can be overwhelming and may require more overhead than it is worth. The distribution of the pad can be challenging, and the sender and receiver must be perfectly synchronized so each is using the same pad.

EXAM TIP The one-time pad, though impractical for most modern applications, is the only perfect cryptosystem.

One-Time Pad Requirements
For a one-time pad encryption scheme to be considered unbreakable, each pad in the scheme must be

- Made up of truly random values
- Used only one time
- Securely distributed to its destination
- Secured at sender's and receiver's sites
- At least as long as the message

Cryptographic Life Cycle

Since most of us will probably not be using one-time pads (the only "perfect" system) to defend our networks, we have to consider that cryptography, like a fine steak, has a limited shelf life. Given enough time and resources, any cryptosystem can be broken, either through analysis or brute force. The *cryptographic life cycle* is the ongoing process of identifying your cryptography needs, selecting the right algorithms, provisioning the needed capabilities and services, and managing keys. Eventually, you determine that your cryptosystem is approaching the end of its shelf life and you start the cycle all over again.

How can you tell when your algorithms (or choice of keyspaces) are about to go stale? You need to stay up to date with the cryptologic research community. They are the best source for early warning that things are going sour. Typically, research papers postulating weaknesses in an algorithm are followed by academic exercises in breaking the algorithm under controlled conditions, which are then followed by articles on how it is broken in general cases. When the first papers come out, it is time to start looking for replacements.

Cryptographic Methods

By far, the most commonly used cryptographic methods today are *symmetric key cryptography*, which uses symmetric keys (also called secret keys), and *asymmetric key cryptography*, which uses two different, or asymmetric, keys (also called public and private keys). Asymmetric key cryptography is also called *public key cryptography* because one of its keys can be made public. As we will see shortly, public key cryptography typically uses powers of prime numbers for encryption and decryption. A variant of this approach uses elliptic curves, which allows much smaller keys to be just as secure and is (unsurprisingly) called *elliptic curve cryptography (ECC)*. Though you may not know it, it is likely that you've used ECC at some point to communicate securely on the Web. (More on that later.) Though these three cryptographic methods are considered secure today (given that you use good keys), the application of quantum computing to cryptology could dramatically change this situation. The following sections explain the key points of these four methods of encryption.

Symmetric Key Cryptography

In a cryptosystem that uses symmetric key cryptography, the sender and receiver use two instances of the same key for encryption and decryption, as shown in Figure 8-6. So the key has dual functionality in that it can carry out both encryption and decryption processes. Symmetric keys are also called *secret* keys, because this type of encryption relies on each user to keep the key a secret and properly protected. If an intruder were to get this key, he could decrypt any intercepted message encrypted with it.

Each pair of users who want to exchange data using symmetric key encryption must have two instances of the same key. This means that if Dan and Iqqi want to communicate, both need to obtain a copy of the same key. If Dan also wants to communicate using symmetric encryption with Norm and Dave, he needs to have three separate keys, one for each friend. This might not sound like a big deal until Dan realizes that he may communicate with hundreds of people over a period of several months, and keeping track and using the correct key that corresponds to each specific receiver can become a daunting task. If 10 people needed to communicate securely with each other using symmetric keys, then 45 keys would need to be kept track of. If 100 people were going to communicate, then 4,950 keys would be involved. The equation used to calculate the number of symmetric keys needed is

$$N(N - 1)/2 = \text{number of keys}$$

The security of the symmetric encryption method is completely dependent on how well users protect their shared keys. This should raise red flags for you if you have ever had to depend on a whole staff of people to keep a secret. If a key is compromised, then all messages encrypted with that key can be decrypted and read by an intruder. This is complicated further by how symmetric keys are actually shared and updated when necessary. If Dan wants to communicate with Norm for the first time, Dan has to figure out how to get the right key to Norm securely. It is not safe to just send it in an e-mail

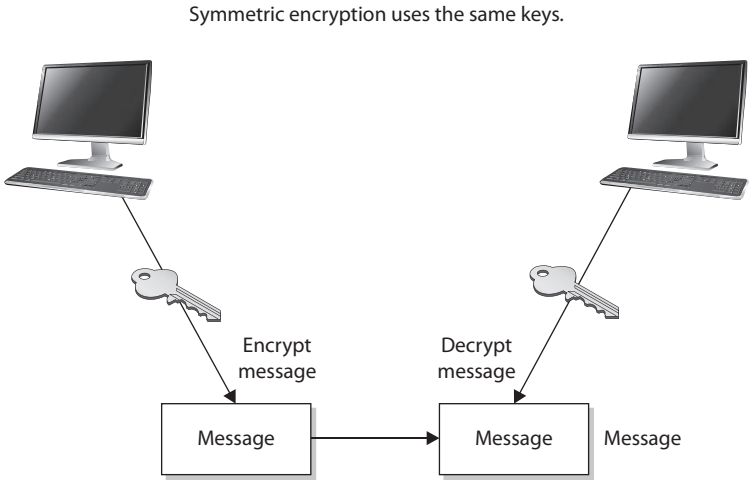


Figure 8-6 When using symmetric algorithms, the sender and receiver use the same key for encryption and decryption functions.

Symmetric Key Cryptosystems Summary

The following outlines the strengths and weaknesses of symmetric key algorithms.

Strengths:

- Much faster (less computationally intensive) than asymmetric systems.
- Hard to break if using a large key size.

Weaknesses:

- Requires a secure mechanism to deliver keys properly.
- Each pair of users needs a unique key, so as the number of individuals increases, so does the number of keys, possibly making key management overwhelming.
- Provides confidentiality but not authenticity or nonrepudiation.

Examples:

- Advanced Encryption Standard (AES)
- ChaCha20

message, because the key is not protected and can be easily intercepted and used by attackers. Thus, Dan must get the key to Norm through an *out-of-band method*. Dan can save the key on a thumb drive and walk over to Norm's desk, or have a secure courier deliver it to Norm. This is a huge hassle, and each method is very clumsy and insecure.

Because both users employ the same key to encrypt and decrypt messages, symmetric cryptosystems can provide confidentiality, but they cannot provide authentication or nonrepudiation. There is no way to prove through cryptography who actually sent a message if two people are using the same key.

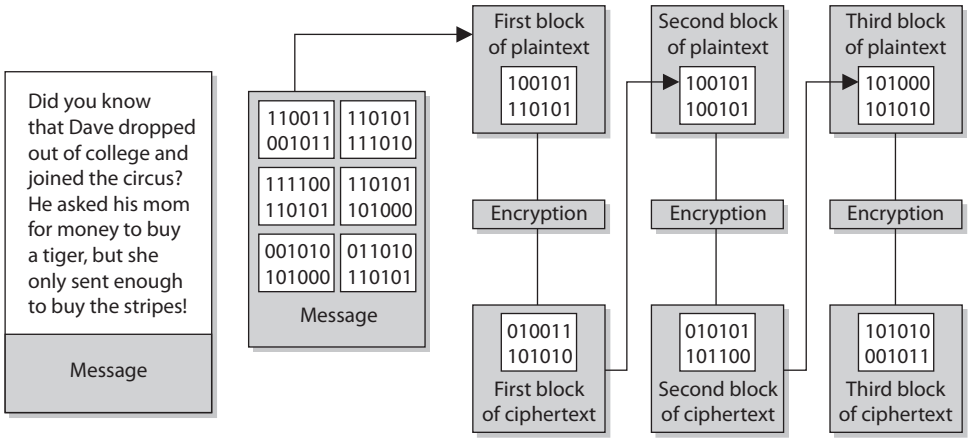
If symmetric cryptosystems have so many problems and flaws, why use them at all? Because they are very fast and can be hard to break. Compared with asymmetric systems, symmetric algorithms scream in speed. They can encrypt and decrypt relatively quickly large amounts of data that would take an unacceptable amount of time to encrypt and decrypt with an asymmetric algorithm. It is also difficult to uncover data encrypted with a symmetric algorithm if a large key size is used. For many of our applications that require encryption, symmetric key cryptography is the only option.

The two main types of symmetric algorithms are block ciphers, which work on blocks of bits, and stream ciphers, which work on one bit at a time.

Block Ciphers

When a *block cipher* is used for encryption and decryption purposes, the message is divided into blocks of bits. These blocks are then put through mathematical functions, one block at a time. Suppose you need to encrypt a message you are sending to your

mother and you are using a block cipher that uses 64 bits. Your message of 640 bits is chopped up into 10 individual blocks of 64 bits. Each block is put through a succession of mathematical formulas, and what you end up with is 10 blocks of encrypted text.



You send this encrypted message to your mother. She has to have the same block cipher and key, and those 10 ciphertext blocks go back through the algorithm in the reverse sequence and end up in your plaintext message.

A strong cipher contains the right level of two main attributes: confusion and diffusion. *Confusion* is commonly carried out through substitution, while *diffusion* is carried out by using transposition. For a cipher to be considered strong, it must contain both of these attributes to ensure that reverse-engineering is basically impossible. The randomness of the key values and the complexity of the mathematical functions dictate the level of confusion and diffusion involved.

In algorithms, diffusion takes place as individual bits of a block are scrambled, or *diffused*, throughout that block. Confusion is provided by carrying out complex substitution functions so the eavesdropper cannot figure out how to substitute the right values and come up with the original plaintext. Suppose you have 500 wooden blocks with individual letters written on them. You line them all up to spell out a paragraph (plaintext). Then you substitute 300 of them with another set of 300 blocks (confusion through substitution). Then you scramble all of these blocks (diffusion through transposition) and leave them in a pile. For someone else to figure out your original message, they would have to substitute the correct blocks and then put them back in the right order. Good luck.

Confusion pertains to making the relationship between the key and resulting ciphertext as complex as possible so the key cannot be uncovered from the ciphertext. Each ciphertext value should depend upon several parts of the key, but this mapping between the key values and the ciphertext values should seem completely random to the observer.

Diffusion, on the other hand, means that a single plaintext bit has influence over several of the ciphertext bits. Changing a plaintext value should change many ciphertext

values, not just one. In fact, in a strong block cipher, if one plaintext bit is changed, it will change every ciphertext bit with the probability of 50 percent. This means that if one plaintext bit changes, then about half of the ciphertext bits will change.

A very similar concept of diffusion is the *avalanche effect*. If an algorithm follows strict avalanche effect criteria, this means that if the input to an algorithm is slightly modified, then the output of the algorithm is changed significantly. So a small change to the key or the plaintext should cause drastic changes to the resulting ciphertext. The ideas of diffusion and avalanche effect are basically the same—they were just derived from different people. Horst Feistel came up with the avalanche term, while Claude Shannon came up with the diffusion term. If an algorithm does not exhibit the necessary degree of the avalanche effect, then the algorithm is using poor randomization. This can make it easier for an attacker to break the algorithm.

Block ciphers use diffusion and confusion in their methods. Figure 8-7 shows a conceptual example of a simplistic block cipher. It has four block inputs, and each block is made up of 4 bits. The block algorithm has two layers of 4-bit substitution boxes called

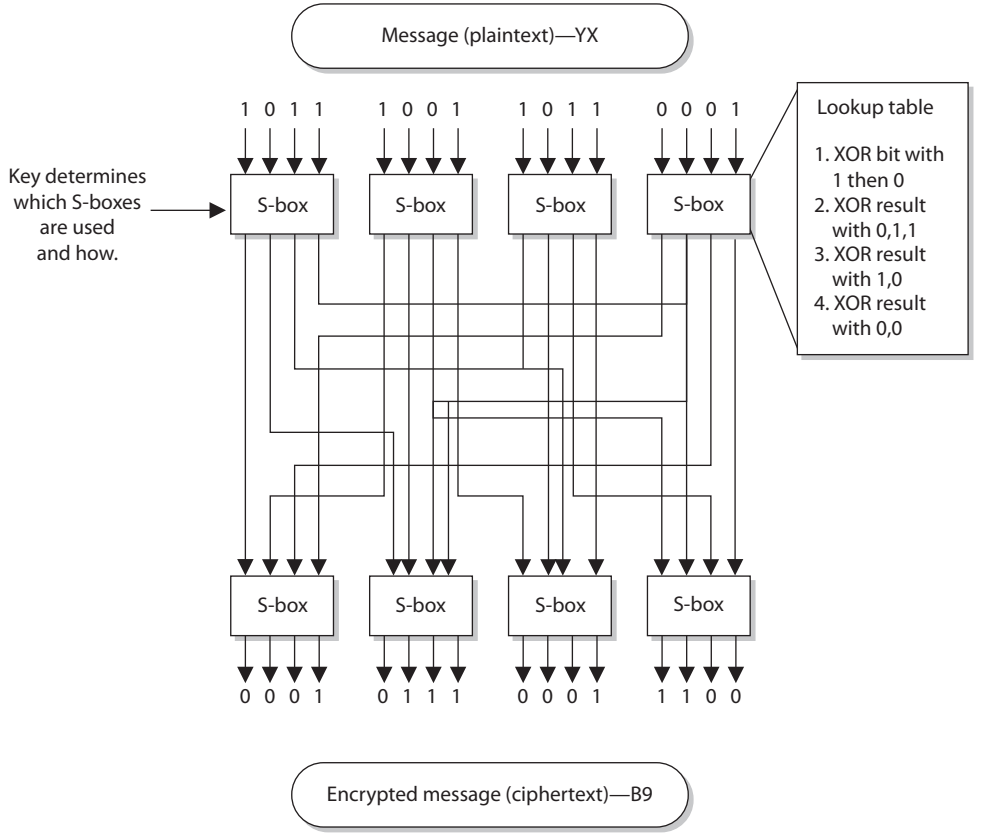


Figure 8-7 A message is divided into blocks of bits, and substitution and transposition functions are performed on those blocks.

S-boxes. Each S-box contains a lookup table used by the algorithm as instructions on how the bits should be encrypted.

Figure 8-7 shows that the key dictates what S-boxes are to be used when scrambling the original message from readable plaintext to encrypted nonreadable ciphertext. Each S-box contains the different substitution methods that can be performed on each block. This example is simplistic—most block ciphers work with blocks of 32, 64, or 128 bits in size, and many more S-boxes are usually involved.

Stream Ciphers

As stated earlier, a block cipher performs mathematical functions on blocks of bits. A stream cipher, on the other hand, does not divide a message into blocks. Instead, a *stream cipher* treats the message as a stream of bits and performs mathematical functions on each bit individually.

When using a stream cipher, a plaintext bit will be transformed into a different ciphertext bit each time it is encrypted. Stream ciphers use *keystream generators*, which produce a stream of bits that is XORed with the plaintext bits to produce ciphertext, as shown in Figure 8-8.

NOTE This process is very similar to the one-time pad explained earlier. The individual bits in the one-time pad are used to encrypt the individual bits of the message through the XOR function, and in a stream algorithm the individual bits created by the keystream generator are used to encrypt the bits of the message through XOR also.

In block ciphers, it is the key that determines what functions are applied to the plaintext and in what order. The key provides the randomness of the encryption process. As stated earlier, most encryption algorithms are public, so people know how they work. The secret to the secret sauce is the key. In stream ciphers, the key also provides randomness, so that the stream of bits that is XORed to the plaintext is as random as possible. This concept

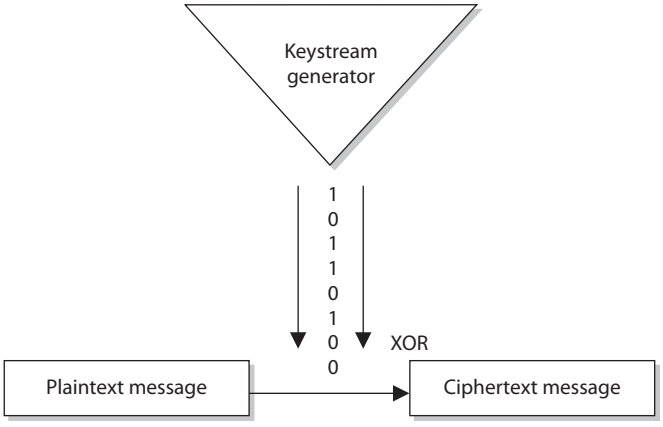


Figure 8-8 With stream ciphers, the bits generated by the keystream generator are XORed with the bits of the plaintext message.

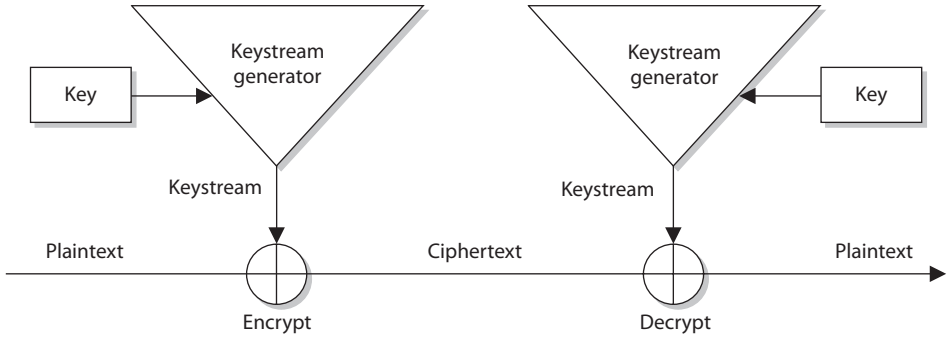


Figure 8-9 The sender and receiver must have the same key to generate the same keystream.

is shown in Figure 8-9. As you can see in this graphic, both the sending and receiving ends must have the same key to generate the same keystream for proper encryption and decryption purposes.

Initialization Vectors

Initialization vectors (IVs) are random values that are used with algorithms to ensure patterns are not created during the encryption process. They are used with keys and do not need to be encrypted when being sent to the destination. If IVs are not used, then two identical plaintext values that are encrypted with the same key will create the same ciphertext. Providing attackers with these types of patterns can make their job easier in breaking the encryption method and uncovering the key. For example, if we have the plaintext value of "See Spot run" two times within our message, we need to make sure that even though there is a pattern in the plaintext message, a pattern in the resulting ciphertext will not be created. So the IV and key are both used by the algorithm to provide more randomness to the encryption process.

A strong and effective stream cipher contains the following characteristics:

- **Easy to implement in hardware** Complexity in the hardware design makes it more difficult to verify the correctness of the implementation and can slow it down.
- **Long periods of no repeating patterns within keystream values** Bits generated by the keystream are not truly random in most cases, which will eventually lead to the emergence of patterns; we want these patterns to be rare.
- **A keystream not linearly related to the key** If someone figures out the keystream values, that does not mean she now knows the key value.
- **Statistically unbiased keystream (as many zeroes as ones)** There should be no dominance in the number of zeroes or ones in the keystream.

Stream ciphers require a lot of randomness and encrypt individual bits at a time. This requires more processing power than block ciphers require, which is why stream ciphers

are better suited to be implemented at the hardware level. Because block ciphers do not require as much processing power, they can be easily implemented at the software level.

Asymmetric Key Cryptography

In symmetric key cryptography, a single secret key is used between entities, whereas in public key systems, each entity has different, *asymmetric keys*. The two different asymmetric keys are mathematically related. If a message is encrypted by one key, the other key is required in order to decrypt the message. One key is called public and the other one private. The *public key* can be known to everyone, and the *private key* must be known and used only by the owner. Many times, public keys are listed in directories and databases of e-mail addresses so they are available to anyone who wants to use these keys to encrypt or decrypt data when communicating with a particular person. Figure 8-10 illustrates the use of the different keys.

The public and private keys of an asymmetric cryptosystem are mathematically related, but if someone gets another person's public key, she should not be able to figure out the corresponding private key. This means that if an eavesdropper gets a copy of Bob's public key, she can't employ some mathematical magic and find out Bob's private key. But if someone gets Bob's private key, then there is big trouble—no one other than the owner should have access to a private key.

If Bob encrypts data with his private key, the receiver must have a copy of Bob's public key to decrypt it. The receiver can decrypt Bob's message and decide to reply to Bob in an encrypted form. All the receiver needs to do is encrypt her reply with Bob's public key, and then Bob can decrypt the message with his private key. It is not possible to encrypt and decrypt using the same key when using an asymmetric key encryption technology because, although mathematically related, the two keys are not the same key, as they are in symmetric cryptography. Bob can encrypt data with his private key, and the receiver can then decrypt it with Bob's public key. By decrypting the message with Bob's

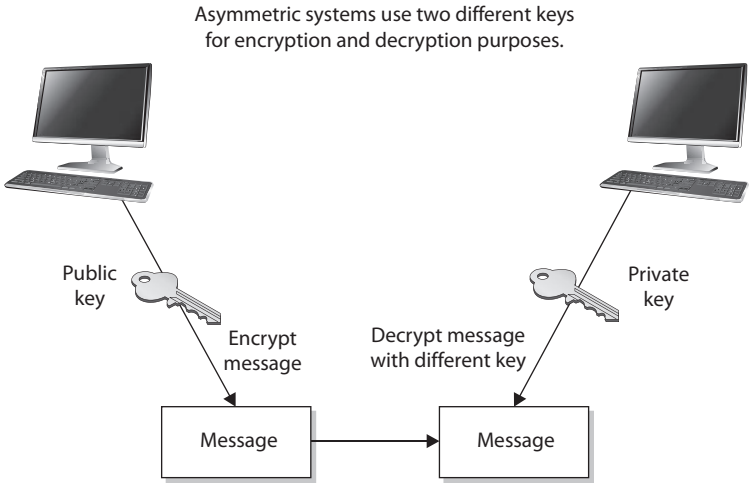


Figure 8-10
An asymmetric cryptosystem

public key, the receiver can be sure the message really came from Bob. A message can be decrypted with a public key only if the message was encrypted with the corresponding private key. This provides authentication, because Bob is the only one who is supposed to have his private key. However, it does not truly provide confidentiality because anyone with the public key (which is, after all, public) can decrypt it. If the receiver wants to make sure Bob is the only one who can read her reply, she will encrypt the response with his public key. Only Bob will be able to decrypt the message because he is the only one who has the necessary private key.

The receiver can also choose to encrypt data with her private key instead of using Bob's public key. Why would she do that? Authentication—she wants Bob to know that the message came from her and no one else. If she encrypted the data with Bob's public key, it does not provide authenticity because anyone can get Bob's public key. If she uses her private key to encrypt the data, then Bob can be sure the message came from her and no one else. Symmetric keys do not provide authenticity, because the same key is used on both ends. Using one of the secret keys does not ensure the message originated from a specific individual.

If confidentiality is the most important security service to a sender, she would encrypt the file with the receiver's public key. This is called a *secure message format* because it can only be decrypted by the person who has the corresponding private key.

If authentication is the most important security service to the sender, then she would encrypt the data with her private key. This provides assurance to the receiver that the only person who could have encrypted the data is the individual who has possession of that private key. If the sender encrypted the data with the receiver's public key, authentication is not provided because this public key is available to anyone.

Encrypting data with the sender's private key is called an *open message format* because anyone with a copy of the corresponding public key can decrypt the message. Confidentiality is not ensured.

Each key type can be used to encrypt and decrypt, so do not get confused and think the public key is only for encryption and the private key is only for decryption. They both have the capability to encrypt and decrypt data. However, if data is encrypted with a private key, it cannot be decrypted with a private key. If data is encrypted with a private key, it must be decrypted with the corresponding public key.

An asymmetric algorithm works much more slowly than a symmetric algorithm, because symmetric algorithms carry out relatively simplistic mathematical functions on the bits during the encryption and decryption processes. They substitute and scramble (transposition) bits, which is not overly difficult or processor intensive. The reason it is hard to break this type of encryption is that the symmetric algorithms carry out this type of functionality over and over again. So a set of bits will go through a long series of being substituted and scrambled.

Asymmetric algorithms are slower than symmetric algorithms because they use much more complex mathematics to carry out their functions, which requires more processing time. Although they are slower, asymmetric algorithms can provide authentication and nonrepudiation, depending on the type of algorithm being used. Asymmetric systems also provide for easier and more manageable key distribution than symmetric systems and do not have the scalability issues of symmetric systems. The reason for these differences

Asymmetric Key Cryptosystems Summary

The following outlines the strengths and weaknesses of asymmetric key algorithms.

Strengths:

- Better key distribution than symmetric systems.
- Better scalability than symmetric systems.
- Can provide authentication and nonrepudiation.

Weaknesses:

- Works much more slowly than symmetric systems.
- Mathematically intensive tasks.

Examples:

- Rivest-Shamir-Adleman (RSA)
- Elliptic curve cryptography (ECC)
- Digital Signature Algorithm (DSA)

is that, with asymmetric systems, you can send out your public key to all of the people you need to communicate with, instead of keeping track of a unique key for each one of them. The "Hybrid Encryption Methods" section later in this chapter shows how these two systems can be used together to get the best of both worlds.

TIP Public key cryptography is asymmetric cryptography. The terms can be used interchangeably.

Table 8-1 summarizes the differences between symmetric and asymmetric algorithms.

Diffie-Hellman Algorithm

The first group to address the shortfalls of symmetric key cryptography decided to attack the issue of secure distribution of the symmetric key. Whitfield Diffie and Martin Hellman worked on this problem and ended up developing the first asymmetric key agreement algorithm, called, naturally, Diffie-Hellman.

To understand how *Diffie-Hellman* works, consider an example. Let's say that Tanya and Erika would like to communicate over an encrypted channel by using Diffie-Hellman. They would both generate a private and public key pair and exchange public keys. Tanya's software would take her private key (which is just a numeric value) and Erika's public key (another numeric value) and put them through the Diffie-Hellman algorithm. Erika's software would take her private key and Tanya's public key and insert them into the Diffie-Hellman algorithm on her computer. Through this process, Tanya and Erika derive the same shared value, which is used to create instances of symmetric keys.

Attribute	Symmetric	Asymmetric
Keys	One key is shared between two or more entities.	One entity has a public key, and the other entity has the corresponding private key.
Key exchange	Out-of-band through secure mechanisms.	A public key is made available to everyone, and a private key is kept secret by the owner.
Speed	The algorithm is less complex and faster.	The algorithm is more complex and slower.
Use	Bulk encryption, which means encrypting files and communication paths.	Key distribution and digital signatures.
Security service provided	Confidentiality.	Confidentiality, authentication, and nonrepudiation.

Table 8-1 Differences Between Symmetric and Asymmetric Systems

So, Tanya and Erika exchanged information that did not need to be protected (their public keys) over an untrusted network, and in turn generated the exact same symmetric key on each system. They both can now use these symmetric keys to encrypt, transmit, and decrypt information as they communicate with each other.

NOTE The preceding example describes key *agreement*, which is different from key *exchange*, the functionality used by the other asymmetric algorithms that will be discussed in this chapter. With key exchange functionality, the sender encrypts the symmetric key with the receiver's public key before transmission.

The Diffie-Hellman algorithm enables two systems to generate a symmetric key securely without requiring a previous relationship or prior arrangements. The algorithm allows for key distribution, but does not provide encryption or digital signature functionality. The algorithm is based on the difficulty of calculating discrete logarithms in a finite field.

The original Diffie-Hellman algorithm is vulnerable to a man-in-the-middle attack, because no authentication occurs before public keys are exchanged. In our example, when Tanya sends her public key to Erika, how does Erika really know it is Tanya's public key? What if Lance spoofed his identity, told Erika he was Tanya, and sent over his key? Erika would accept this key, thinking it came from Tanya. Let's walk through the steps of how this type of attack would take place, as illustrated in Figure 8-11:

1. Tanya sends her public key to Erika, but Lance grabs the key during transmission so it never makes it to Erika.
2. Lance spoofs Tanya's identity and sends over his public key to Erika. Erika now thinks she has Tanya's public key.

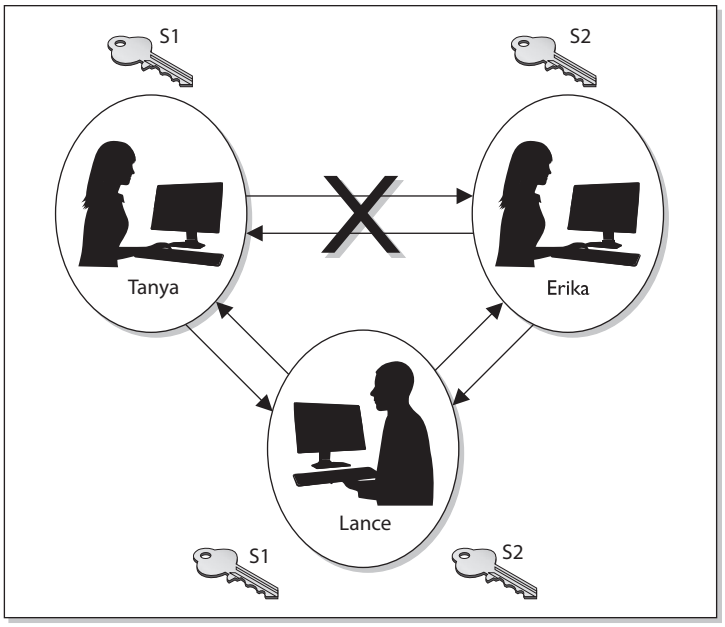


Figure 8-11
A man-in-the-middle attack against a Diffie-Hellman key agreement

3. Erika sends her public key to Tanya, but Lance grabs the key during transmission so it never makes it to Tanya.
4. Lance spoofs Erika's identity and sends over his public key to Tanya. Tanya now thinks she has Erika's public key.
5. Tanya combines her private key and Lance's public key and creates symmetric key S1.
6. Lance combines his private key and Tanya's public key and creates symmetric key S1.
7. Erika combines her private key and Lance's public key and creates symmetric key S2.
8. Lance combines his private key and Erika's public key and creates symmetric key S2.
9. Now Tanya and Lance share a symmetric key (S1) and Erika and Lance share a different symmetric key (S2). Tanya and Erika think they are sharing a key between themselves and do not realize Lance is involved.
10. Tanya writes a message to Erika, uses her symmetric key (S1) to encrypt the message, and sends it.
11. Lance grabs the message and decrypts it with symmetric key S1, reads or modifies the message and re-encrypts it with symmetric key S2, and then sends it to Erika.
12. Erika takes symmetric key S2 and uses it to decrypt and read the message.

The countermeasure to this type of attack is to have authentication take place before accepting someone's public key. The basic idea is that we use some sort of certificate to attest the identity of the party on the other side before trusting the data we receive from it. One of the most common ways to do this authentication is through the use of the RSA cryptosystem, which we describe next.

RSA

RSA, named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman, is a public key algorithm that is the most popular when it comes to asymmetric algorithms. RSA is a worldwide de facto standard and can be used for digital signatures, key exchange, and encryption. It was developed in 1978 at MIT and provides authentication as well as key encryption.

The security of this algorithm comes from the difficulty of factoring large numbers into their original prime numbers. The public and private keys are functions of a pair of large prime numbers, and the necessary activity required to decrypt a message from ciphertext to plaintext using a private key is comparable to factoring a product into two prime numbers.

NOTE A prime number is a positive whole number whose only factors (i.e., integer divisors) are 1 and the number itself.

One advantage of using RSA is that it can be used for encryption and digital signatures. Using its one-way function, RSA provides encryption and signature verification, and the inverse direction performs decryption and signature generation.

RSA has been implemented in applications; in operating systems; and at the hardware level in network interface cards, secure telephones, and smart cards. RSA can be used as a *key exchange protocol*, meaning it is used to encrypt the symmetric key to get it securely to its destination. RSA has been most commonly used with the symmetric algorithm AES. So, when RSA is used as a key exchange protocol, a cryptosystem generates a symmetric key to be used with the AES algorithm. Then the system encrypts the symmetric key with the receiver's public key and sends it to the receiver. The symmetric key is protected because only the individual with the corresponding private key can decrypt and extract the symmetric key.

Diving into Numbers Cryptography is really all about using mathematics to scramble bits into an undecipherable form and then using the same mathematics in reverse to put the bits back into a form that can be understood by computers and people. RSA's mathematics are based on the difficulty of factoring a large integer into its two prime factors. Put on your nerdy hat with the propeller and let's look at how this algorithm works.

The algorithm creates a public key and a private key from a function of large prime numbers. When data is encrypted with a public key, only the corresponding private key can decrypt the data. This act of decryption is basically the same as factoring the product of two prime numbers. So, let's say Ken has a secret (encrypted message), and for you to

be able to uncover the secret, you have to take a specific large number and factor it and come up with the two numbers Ken has written down on a piece of paper. This may sound simplistic, but the number you must properly factor can be 2^{2048} in size. Not as easy as you may think.

The following sequence describes how the RSA algorithm comes up with the keys in the first place:

1. Choose two random large prime numbers, p and q.
2. Generate the product of these numbers: $n = pq$. n is used as the modulus.
3. Choose a random integer e (the public key) that is greater than 1 but less than $(p - 1)(q - 1)$. Make sure that e and $(p - 1)(q - 1)$ are relatively prime.
4. Compute the corresponding private key, d, such that $de - 1$ is a multiple of $(p - 1)(q - 1)$.
5. The public key = (n, e).
6. The private key = (n, d).
7. The original prime numbers p and q are discarded securely.

We now have our public and private keys, but how do they work together?

If someone needs to encrypt message m with your public key (e, n), the following formula results in ciphertext c:

$$c = m^e \bmod n$$

Then you need to decrypt the message with your private key (d), so the following formula is carried out:

$$m = c^d \bmod n$$

In essence, you encrypt a plaintext message by multiplying it by itself e times (taking the modulus, of course), and you decrypt it by multiplying the ciphertext by itself d times (again, taking the modulus). As long as e and d are large enough values, an attacker will have to spend an awfully long time trying to figure out through trial and error the value of d. (Recall that we publish the value of e for the whole world to know.)

You may be thinking, "Well, I don't understand these formulas, but they look simple enough. Why couldn't someone break these small formulas and uncover the encryption key?" Maybe someone will one day. As the human race advances in its understanding of mathematics and as processing power increases and cryptanalysis evolves, the RSA algorithm may be broken one day. If we were to figure out how to quickly and more easily factor large numbers into their original prime values, all of these cards would fall down, and this algorithm would no longer provide the security it does today. But we have not hit that bump in the road yet, so we are all happily using RSA in our computing activities.

One-Way Functions A *one-way function* is a mathematical function that is easier to compute in one direction than in the opposite direction. An analogy of this is when you

drop a glass on the floor. Although dropping a glass on the floor is easy, putting all the pieces back together again to reconstruct the original glass is next to impossible. This concept is similar to how a one-way function is used in cryptography, which is what the RSA algorithm, and all other asymmetric algorithms, are based upon.

The easy direction of computation in the one-way function that is used in the RSA algorithm is the process of multiplying two large prime numbers. If I asked you to multiply two prime numbers, say 79 and 73, it would take you just a few seconds to punch that into a calculator and come up with the product (5,767). Easy. Now, suppose I asked you to find out which two numbers, when multiplied together, produce the value 5,767. This is called factoring and, when the factors involved are large prime numbers, it turns out to be a *really* hard problem. This difficulty in factoring the product of large prime numbers is what provides security for RSA key pairs.

As explained earlier in this chapter, *work factor* is the amount of time and resources it would take for someone to break an encryption method. In asymmetric algorithms, the work factor relates to the difference in time and effort that carrying out a one-way function in the easy direction takes compared to carrying out a one-way function in the hard direction. In most cases, the larger the key size, the longer it would take for the adversary to carry out the one-way function in the hard direction (decrypt a message).

The crux of this section is that all asymmetric algorithms provide security by using mathematical equations that are easy to perform in one direction and next to impossible to perform in the other direction. The "hard" direction is based on a "hard" mathematical problem. RSA's hard mathematical problem requires factoring large numbers into their original prime numbers.

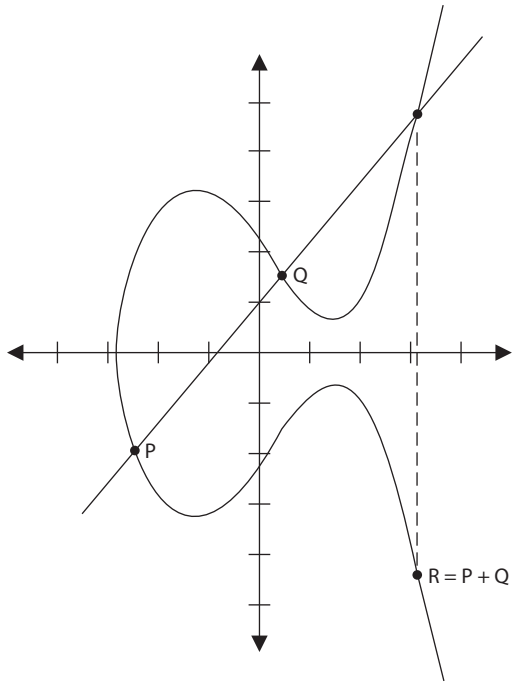
Elliptic Curve Cryptography

The one-way function in RSA has survived cryptanalysis for over four decades but eventually will be cracked simply because we keep building computers that are faster. Sooner or later, computers will be able to factor the products of ever-larger prime numbers in reasonable times, at which point we would need to either ditch RSA or figure out how to use larger keys. Anticipating this eventuality, cryptographers found an even better trapdoor in elliptic curves. An *elliptic curve*, such as the one shown in Figure 8-12, is the set of points that satisfies a specific mathematical equation such as this one:

$$y^2 = x^3 + ax + b$$

Elliptic curves have two properties that are useful for cryptography. The first is that they are symmetrical about the X axis. This means that the top and bottom parts of the curve are mirror images of each other. The second useful property is that a straight line will intersect them in no more than three points. With these properties in mind, you can define a "dot" function that, given two points on the curve, gives you a third point on the flip side of it. Figure 8-12 shows how P dot Q = R. You simply follow the line through P and Q to find its third point of intersection on the curve (which could be between the two), and then drop down to that point R on the mirror image (in this case) below the X axis. You can keep going from there, so R dot P gives you another point that is

Figure 8-12
Elliptic curve



somewhere to the left and up from Q on the curve. If you keep "dotting" the original point P with the result of the previous "dot" operation *n* times (for some reasonably large value of *n*), you end up with a point that is really hard for anyone to guess or brute-force if they don't know the value of *n*. If you do know that value, then computing the final point is pretty easy. That is what makes this a great one-way function.

An *elliptic curve cryptosystem (ECC)* is a public key cryptosystem that can be described by a prime number (the equivalent of the modulus value in RSA), a curve equation, and a public point on the curve. The private key is some number *d*, and the corresponding public key *e* is the public point on the elliptic curve "dotted" with itself *d* times. Computing the private key from the public key in this kind of cryptosystem (i.e., reversing the one-way function) requires calculating the elliptic curve discrete logarithm function, which turns out to be really, really hard.

ECC provides much of the same functionality RSA provides: digital signatures, secure key distribution, and encryption. One differing factor is ECC's efficiency. ECC is more efficient than RSA and any other asymmetric algorithm. To illustrate this, an ECC key of 256 bits offers the equivalent protection of an RSA key of 3,072 bits. This is particularly useful because some devices have limited processing capacity, storage, power supply, and bandwidth, such as wireless devices and mobile telephones. With these types of devices, efficiency of resource use is very important. ECC provides encryption functionality, requiring a smaller percentage of the resources compared to RSA and other algorithms, so it is used in these types of devices.

Quantum Cryptography

Both RSA and ECC rely on the difficulty of reversing one-way functions. But what if we were able to come up with a cryptosystem in which it was impossible (not just difficult) to do this? This is the promise of quantum cryptography, which, despite all the hype, is still very much in its infancy. *Quantum cryptography* is the field of scientific study that applies quantum mechanics to perform cryptographic functions. The most promising application of this field, and the one we may be able to use soonest, provides a solution to the key distribution problem associated with symmetric key cryptosystems.

Quantum key distribution (QKD) is a system that generates and securely distributes encryption keys of any length between two parties. Though we could, in principle, use anything that obeys the principles of quantum mechanics, photons (the tiny particles that make up light) are the most convenient particles to use for QKD. It turns out photons are polarized or spin in ways that can be described as vertical, horizontal, diagonal left (–45°), and diagonal right (45°). If we put a polarized filter in front of a detector, any photon that makes it to that detector will have the polarization of its filter. Two types of filters are commonly used in QKD. The first is rectilinear and allows vertically and horizontally polarized photons through. The other is a (you guessed it) diagonal filter, which allows both diagonally left and diagonally right polarized photons through. It is important to note that the only way to measure the polarization on a photon is to essentially destroy it: either it is blocked by the filter if the polarizations are different or it is absorbed by the sensor if it makes it through.

Let's suppose that Alice wants to securely send an encryption key to Bob using QKD. They would use the following process.

1. They agree beforehand that photons that have either vertical or diagonal-right polarization represent the number zero and those with horizontal or diagonal-left polarization represent the number one.
2. The polarization of each photon is then generated randomly but is known to Alice.
3. Since Bob doesn't know what the correct spins are, he'll pass them through filters, randomly detect the polarization for each photon, and record his results. Because he's just guessing the polarizations, on average, he'll get half of them wrong, as we can see in Figure 8-13. He will, however, know which filter he applied to each photon, whether he got it right or wrong.
4. Once Alice is done sending bits, Bob will send her a message over an insecure channel (they don't need encryption for this), telling her the sequence of polarizations he recorded.
5. Alice will compare Bob's sequence to the correct sequence and tell him which polarizations he got right and which ones he got wrong.
6. They both discard Bob's wrong guesses and keep the remaining sequence of bits. They now have a shared secret key through this process, which is known as *key distillation*.

But what if there's a third, malicious, party eavesdropping on the exchange? Suppose this is Eve and she wants to sniff the secret key so she can intercept whatever messages

Alice's bit	0	1	1	0	1	0	0	1
Alice's basis	+	+	×	+	×	×	×	+
Alice's polarization	↑	→	↖	↑	↖	↗	↗	→
Bob's filter	+	×	×	×	+	×	+	+
Bob's measurement	↑	↗	↖	↗	→	↗	→	→
Shared secret key	0		1			0		1

Figure 8-13 Key distillation between Alice and Bob

Alice and Bob encrypt with it. Since the quantum state of photons is destroyed when they are filtered or measured, she would have to follow the same process as Bob intends to and then generate a new photon stream to forward to Bob. The catch is that Eve (just like Bob) will get 50 percent of the measurements wrong, but (unlike Bob) now has to guess what random basis was used and send these guesses to Bob. When Alice and Bob compare polarizations, they'll note a much higher error rate than normal and be able to infer that someone was eavesdropping.

If you're still awake and paying attention, you may be wondering, "Why use the polarization filters in the first place? Why not just capture the photon and see how it's spinning?" The answer gets complicated in a hurry, but the short version is that polarization is a random quantum state until you pass the photon through the filter and force the photon to "decide" between the two polarizations. Eve cannot just re-create the photon's quantum state like she would do with conventional data. Keep in mind that quantum mechanics are pretty weird but lead to unconditional security of the shared key.

Now that we have a basic idea of how QKD works, let's think back to our discussion of the only perfect and unbreakable cryptosystem: the one-time pad. You may recall that it has five major requirements that largely make it impractical. We list these here and show how QKD addresses each of them rather nicely:

- **Made up of truly random values** Quantum mechanics deals with attributes of matter and energy that are truly random, unlike the pseudo-random numbers we can generate algorithmically on a traditional computer.
- **Used only one time** Since QKD solves the key distribution problem, it allows us to transmit as many unique keys as we want, reducing the temptation (or need) to reuse keys.
- **Securely distributed to its destination** If someone attempts to eavesdrop on the key exchange, they will have to do so actively in a way that, as we've seen, is pretty much guaranteed to produce evidence of their tampering.
- **Secured at sender's and receiver's sites** OK, this one is not really addressed by QKD directly, but anyone going through all this effort would presumably not mess this one up, right?
- **At least as long as the message** Since QKD can be used for arbitrarily long key streams, we can easily generate keys that are at least as long as the longest message we'd like to send.

Now, before you get all excited and try to buy a QKD system for your organization, keep in mind that this technology is not quite ready for prime time. To be clear, commercial QKD devices are available as a "plug and play" option. Some banks in Geneva, Switzerland, use QKD to secure bank-to-bank traffic, and the Canton of Geneva uses it to secure online voting. The biggest challenge to widespread adoption of QKD at this point is the limitation on the distance at which photons can be reliably transmitted. As we write these lines, the maximum range for QKD is just over 500 km over fiber-optic wires. While space-to-ground QKD has been demonstrated using satellites and ground stations, drastically increasing the reach of such systems, it remains extremely difficult due to atmospheric interference. Once this problem is solved, we should be able to leverage a global, satellite-based QKD network.

Hybrid Encryption Methods

Up to this point, we have figured out that symmetric algorithms are fast but have some drawbacks (lack of scalability, difficult key management, and provide only confidentiality). Asymmetric algorithms do not have these drawbacks but are very slow. We just can't seem to win. So we turn to a hybrid system that uses symmetric and asymmetric encryption methods together.

Asymmetric and Symmetric Algorithms Used Together

Asymmetric and symmetric cryptosystems are used together very frequently. In this hybrid approach, the two technologies are used in a complementary manner, with each performing a different function. A symmetric algorithm creates keys used for encrypting bulk data, and an asymmetric algorithm creates keys used for automated key distribution. Each algorithm has its pros and cons, so using them together can be the best of both worlds.

When a symmetric key is used for bulk data encryption, this key is used to encrypt the message you want to send. When your friend gets the message you encrypted, you want him to be able to decrypt it, so you need to send him the necessary symmetric key to use to decrypt the message. You do not want this key to travel unprotected, because if the message were intercepted and the key were not protected, an eavesdropper could intercept the message that contains the necessary key to decrypt your message and read your information. If the symmetric key needed to decrypt your message is not protected, there is no use in encrypting the message in the first place. So you should use an asymmetric algorithm to encrypt the symmetric key, as depicted in Figure 8-14. Why use the symmetric key on the message and the asymmetric key on the symmetric key? As stated earlier, the asymmetric algorithm takes longer because the math is more complex. Because your message is most likely going to be longer than the length of the key, you use the faster algorithm (symmetric) on the message and the slower algorithm (asymmetric) on the key.

How does this actually work? Let's say Bill is sending Paul a message that Bill wants only Paul to be able to read. Bill encrypts his message with a secret key, so now Bill has ciphertext and a symmetric key. The key needs to be protected, so Bill encrypts the symmetric key with an asymmetric key. Remember that asymmetric algorithms use private and public keys, so Bill will encrypt the symmetric key with Paul's public key. Now Bill has ciphertext from the message and ciphertext from the symmetric key. Why did Bill encrypt the symmetric key with Paul's public key instead of his own private key? Because if Bill

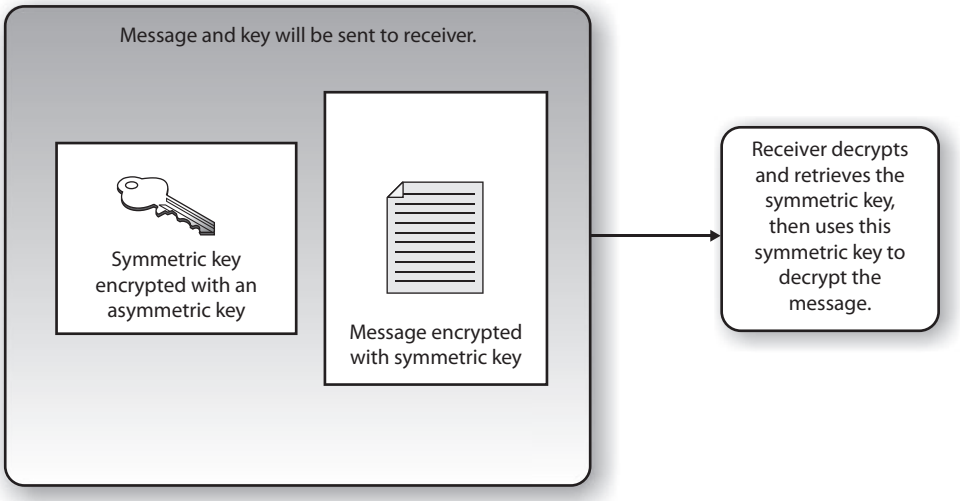
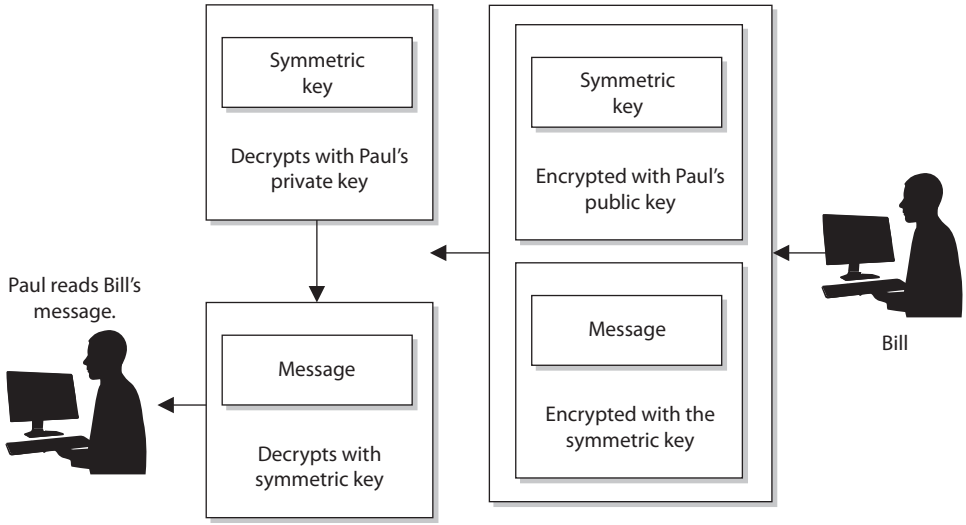


Figure 8-14 In a hybrid system, the asymmetric key is used to encrypt the symmetric key, and the symmetric key is used to encrypt the message

encrypted it with his own private key, then anyone with Bill's public key could decrypt it and retrieve the symmetric key. However, Bill does not want anyone who has his public key to read his message to Paul. Bill only wants Paul to be able to read it. So Bill encrypts the symmetric key with Paul's public key. If Paul has done a good job protecting his private key, he will be the only one who can read Bill's message.

Paul receives Bill's message, and Paul uses his private key to decrypt the symmetric key. Paul then uses the symmetric key to decrypt the message. Paul then reads Bill's very important and confidential message that asks Paul how his day is.



Now, when we say that Bill is using this key to encrypt and that Paul is using that key to decrypt, those two individuals do not necessarily need to find the key on their hard drive and know how to properly apply it. We have software to do this for us—thank goodness.

If this is your first time with these issues and you are struggling, don't worry. Just remember the following points:

- An asymmetric algorithm performs encryption and decryption by using public and private keys that are related to each other mathematically.
- A symmetric algorithm performs encryption and decryption by using a shared secret key.
- A symmetric key is used to encrypt and/or decrypt the actual message.
- Public keys are used to encrypt the symmetric key for secure key exchange.
- A secret key is synonymous with a symmetric key.
- An asymmetric key refers to a public or private key.

So, that is how a hybrid system works. The symmetric algorithm uses a secret key that will be used to encrypt the bulk, or the message, and the asymmetric key encrypts the secret key for transmission.

To ensure that some of these concepts are driven home, ask these questions of yourself without reading the answers provided:

1. If a symmetric key is encrypted with a receiver's public key, what security service(s) is (are) provided?
2. If data is encrypted with the sender's private key, what security service(s) is (are) provided?
3. If the sender encrypts data with the receiver's private key, what security services(s) is (are) provided?
4. Why do we encrypt the message with the symmetric key?
5. Why don't we encrypt the symmetric key with another symmetric key?

Now check your answers:

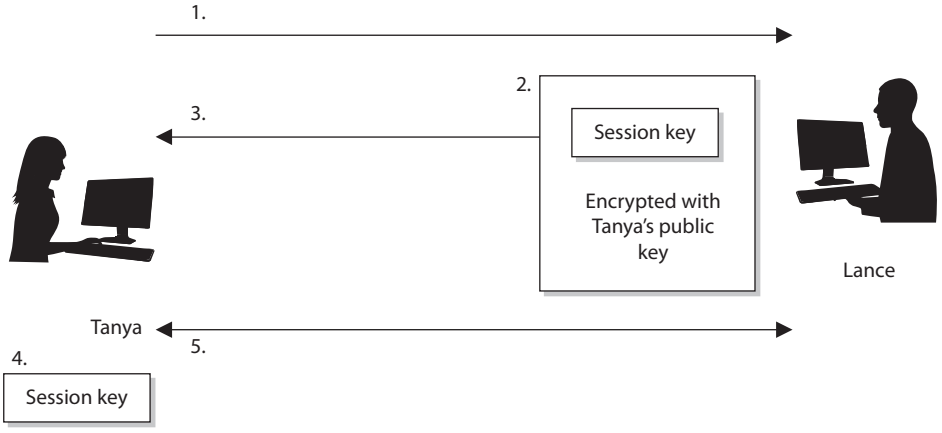
1. Confidentiality, because only the receiver's private key can be used to decrypt the symmetric key, and only the receiver should have access to this private key.
2. Authenticity of the sender and nonrepudiation. If the receiver can decrypt the encrypted data with the sender's public key, then she knows the data was encrypted with the sender's private key.

3. None, because no one but the owner of the private key should have access to it. Trick question.
4. Because the asymmetric key algorithm is too slow.
5. We need to get the necessary symmetric key to the destination securely, which can only be carried out through asymmetric cryptography via the use of public and private keys to provide a mechanism for secure transport of the symmetric key.

Session Keys

A *session key* is a single-use symmetric key that is used to encrypt messages between two users during a communication session. A session key is no different from the symmetric key described in the previous section, but it is only good for one communication session between users.

If Tanya has a symmetric key she uses to always encrypt messages between Lance and herself, then this symmetric key would not be regenerated or changed. They would use the same key every time they communicated using encryption. However, using the same key repeatedly increases the chances of the key being captured and the secure communication being compromised. If, on the other hand, a new symmetric key were generated each time Lance and Tanya wanted to communicate, as shown in Figure 8-15, it would be used only during their one dialogue and then destroyed. If they wanted to communicate an hour later, a new session key would be created and shared.



1) Tanya sends Lance her public key.
2) Lance generates a random session key and encrypts it using Tanya's public key.
3) Lance sends the session key, encrypted with Tanya's public key, to Tanya.
4) Tanya decrypts Lance's message with her private key and now has a copy of the session key.
5) Tanya and Lance use this session key to encrypt and decrypt messages to each other.

Figure 8-15 A session key is generated so all messages can be encrypted during one particular session between users.

A session key provides more protection than static symmetric keys because it is valid for only one session between two computers. If an attacker were able to capture the session key, she would have a very small window of time to use it to try to decrypt messages being passed back and forth.

In cryptography, almost all data encryption takes place through the use of session keys. When you write an e-mail and encrypt it before sending it over the wire, it is actually being encrypted with a session key. If you write another message to the same person one minute later, a brand-new session key is created to encrypt that new message. So if an eavesdropper happens to figure out one session key, that does not mean she has access to all other messages you write and send off.

When two computers want to communicate using encryption, they must first go through a handshaking process. The two computers agree on the encryption algorithms that will be used and exchange the session key that will be used for data encryption. In a sense, the two computers set up a virtual connection between each other and are said to be in session. When this session is done, each computer tears down any data structures it built to enable this communication to take place, releases the resources, and destroys the session key. These things are taken care of by operating systems and applications in the background, so a user would not necessarily need to be worried about using the wrong type of key for the wrong reason. The software will handle this, but it is important for security professionals to understand the difference between the key types and the issues that surround them.

CAUTION Private and symmetric keys should not be available in cleartext. This may seem obvious to you, but there have been several implementations over time that have allowed for this type of compromise to take place.

Unfortunately, we don't always seem to be able to call an apple an apple. In many types of technology, the exact same thing can have more than one name. For example, symmetric cryptography can be referred to as any of the following:

- Secret key cryptography
- Session key cryptography
- Shared key cryptography
- Private key cryptography

We know the difference between secret keys (static) and session keys (dynamic), but what is this "shared key" and "private key" mess? Well, using the term "shared key" makes sense, because the sender and receiver are sharing one single key. It's unfortunate that the term "private key" can be used to describe symmetric cryptography, because it only adds more confusion to the difference between symmetric cryptography (where one symmetric key is used) and asymmetric cryptography (where both a private and public key are used). You just need to remember this little quirk and still understand the difference between symmetric and asymmetric cryptography.

Integrity

Cryptography is mainly concerned with protecting the confidentiality of information. It can also, however, allow us to ensure its integrity. In other words, how can we be certain that a message we receive or a file we download has not been modified? For this type of protection, hash algorithms are required to successfully detect intentional and unintentional unauthorized modifications to data. However, as we will see shortly, it is possible for attackers to modify data, recompute the hash, and deceive the recipient. In some cases, we need a more robust approach to message integrity verification. Let's start off with hash algorithms and their characteristics.

Hashing Functions

A *one-way hash* is a function that takes a variable-length string (a message) and produces a fixed-length value called a *hash value*. For example, if Kevin wants to send a message to Maureen and he wants to ensure the message does not get altered in an unauthorized fashion while it is being transmitted, he would calculate a hash value for the message and append it to the message itself. When Maureen receives the message, she performs the same hashing function Kevin used and then compares her result with the hash value sent with the message. If the two values are the same, Maureen can be sure the message was not altered during transmission. If the two values are different, Maureen knows the message was altered, either intentionally or unintentionally, and she discards the message.

The hashing algorithm is not a secret—it is publicly known. The secrecy of the one-way hashing function is its "one-wayness." The function is run in only one direction, not the other direction. This is different from the one-way function used in public key cryptography, in which security is provided based on the fact that, without knowing a trapdoor, it is very hard to perform the one-way function backward on a message and come up with readable plaintext. However, one-way hash functions are never used in reverse; they create a hash value and call it a day. The receiver does not attempt to reverse the process at the other end, but instead runs the same hashing function one way and compares the two results.

EXAM TIP Keep in mind that hashing is not the same thing as encryption; you can't "decrypt" a hash. You can only run the same hashing algorithm against the same piece of text in an attempt to derive the same hash or fingerprint of the text.

Various Hashing Algorithms

As stated earlier, the goal of using a one-way hash function is to provide a fingerprint of the message. If two different messages produce the same hash value, it would be easier for an attacker to break that security mechanism because patterns would be revealed.

A strong one-hash function should not provide the same hash value for two or more different messages. If a hashing algorithm takes steps to ensure it does not create the same hash value for two or more messages, it is said to be *collision free*.

Algorithm	Description
Message Digest 5 (MD5) algorithm	Produces a 128-bit hash value. More complex than MD4.
Secure Hash Algorithm (SHA)	Produces a 160-bit hash value. Used with Digital Signature Algorithm (DSA).
SHA-1, SHA-256, SHA-384, SHA-512	Updated versions of SHA. SHA-1 produces a 160-bit hash value, SHA-256 creates a 256-bit value, and so on.

Table 8-2 Various Hashing Algorithms Available

Strong cryptographic hash functions have the following characteristics:

- The hash should be computed over the entire message.
- The hash should be a one-way function so messages are not disclosed by their values.
- Given a message and its hash value, computing another message with the same hash value should be impossible.
- The function should be resistant to birthday attacks (explained in the upcoming section "Attacks Against One-Way Hash Functions").

Table 8-2 and the following sections quickly describe some of the available hashing algorithms used in cryptography today.

MD5 *MD5* was created by Ron Rivest in 1991 as a better version of his previous message digest algorithm (MD4). It produces a 128-bit hash, but the algorithm is subject to collision attacks, and is therefore no longer suitable for applications like digital certificates and signatures that require collision attack resistance. It is still commonly used for file integrity checksums, such as those required by some intrusion detection systems, as well as for forensic evidence integrity.

SHA *SHA* was designed by the NSA and published by the National Institute of Standards and Technology (NIST) to be used with the Digital Signature Standard (DSS), which is discussed a bit later in more depth. SHA was designed to be used in digital signatures and was developed when a more secure hashing algorithm was required for U.S. government applications. It produces a 160-bit hash value, or message digest. This is then inputted into an asymmetric algorithm, which computes the signature for a message.

SHA is similar to MD5. It has some extra mathematical functions and produces a 160-bit hash instead of a 128-bit hash, which initially made it more resistant to collision attacks. Newer versions of this algorithm (collectively known as the SHA-2 and SHA-3 families) have been developed and released: SHA-256, SHA-384, and SHA-512. The SHA-2 and SHA-3 families are considered secure for all uses.

Attacks Against One-Way Hash Functions

A strong hashing algorithm does not produce the same hash value for two different messages. If the algorithm does produce the same value for two distinctly different messages, this is called a *collision*. An attacker can attempt to force a collision, which is referred to as a *birthday attack*. This attack is based on the mathematical birthday paradox that exists in standard statistics. Now hold on to your hat while we go through this—it is a bit tricky:

> How many people must be in the same room for the chance to be greater than even that another person has the same birthday as you?
> **Answer:** 253
>
> How many people must be in the same room for the chance to be greater than even that at least two people share the same birthday?
> **Answer:** 23

This seems a bit backward, but the difference is that in the first instance, you are looking for someone with a specific birthday date that matches yours. In the second instance, you are looking for any two people who share the same birthday. There is a higher probability of finding two people who share a birthday than of finding another person who shares your birthday. Or, stated another way, it is easier to find two matching values in a sea of values than to find a match for just one specific value.

Why do we care? The birthday paradox can apply to cryptography as well. Since any random set of 23 people most likely (at least a 50 percent chance) includes two people who share a birthday, by extension, if a hashing algorithm generates a message digest of 60 bits, there is a high likelihood that an adversary can find a collision using only 2^{30} inputs.

The main way an attacker can find the corresponding hashing value that matches a specific message is through a brute-force attack. If he finds a message with a specific hash value, it is equivalent to finding someone with a specific birthday. If he finds two messages with the same hash values, it is equivalent to finding two people with the same birthday.

The output of a hashing algorithm is n, and to find a message through a brute-force attack that results in a specific hash value would require hashing $2n$ random messages. To take this one step further, finding two messages that hash to the same value would require review of only $2n/2$ messages.

How Would a Birthday Attack Take Place?

Sue and Joe are going to get married, but before they do, they have a prenuptial contract drawn up that states if they get divorced, then Sue takes her original belongings and Joe takes his original belongings. To ensure this contract is not modified, it is hashed and a message digest value is created.

One month after Sue and Joe get married, Sue carries out some devious activity behind Joe's back. She makes a copy of the message digest value without anyone knowing. Then she makes a new contract that states that if Joe and Sue get a divorce, Sue owns both her

own original belongings and Joe's original belongings. Sue hashes this new contract and compares the new message digest value with the message digest value that correlates with the contract. They don't match. So Sue tweaks her contract ever so slightly and creates another message digest value and compares them. She continues to tweak her contract until she forces a collision, meaning her contract creates the same message digest value as the original contract. Sue then changes out the original contract with her new contract and quickly divorces Joe. When Sue goes to collect Joe's belongings and he objects, she shows him that no modification could have taken place on the original document because it still hashes out to the same message digest. Sue then moves to an island.

Hash algorithms usually use message digest sizes (the value of *n*) that are large enough to make collisions difficult to accomplish, but they are still possible. An algorithm that has 256-bit output, like SHA-256, may require approximately 2^{128} computations to break. This means there is a less than 1 in 2^{128} chance that someone could carry out a successful birthday attack.

The main point of discussing this paradox is to show how important longer hashing values truly are. A hashing algorithm that has a larger bit output is less vulnerable to brute-force attacks such as a birthday attack. This is the primary reason why the new versions of SHA have such large message digest values.

Message Integrity Verification

Whether messages are encrypted or not, we frequently want to ensure that they arrive at their destination with no alterations, accidental or deliberate. We can use the principles we've discussed in this chapter to ensure the integrity of our traffic to various degrees of security. Let's look at three increasingly more powerful ways to do this, starting with a simple message digest.

Message Digest

A one-way hashing function takes place without the use of any keys. This means, for example, that if Cheryl writes a message, calculates a message digest, appends the digest to the message, and sends it on to Scott, Bruce can intercept this message, alter Cheryl's message, recalculate another message digest, append it to the message, and send it on to Scott. When Scott receives it, he verifies the message digest, but never knows the message was actually altered by Bruce. Scott thinks the message came straight from Cheryl and was never modified because the two message digest values are the same. This process is depicted in Figure 8-16 and consists of the following steps:

1. The sender writes a message.
2. The sender puts the message through a hashing function, generating a message digest.
3. The sender appends the message digest to the message and sends it to the receiver.
4. The receiver puts the message through a hashing function and generates his own message digest.
5. The receiver compares the two message digest values. If they are the same, the message has not been altered.

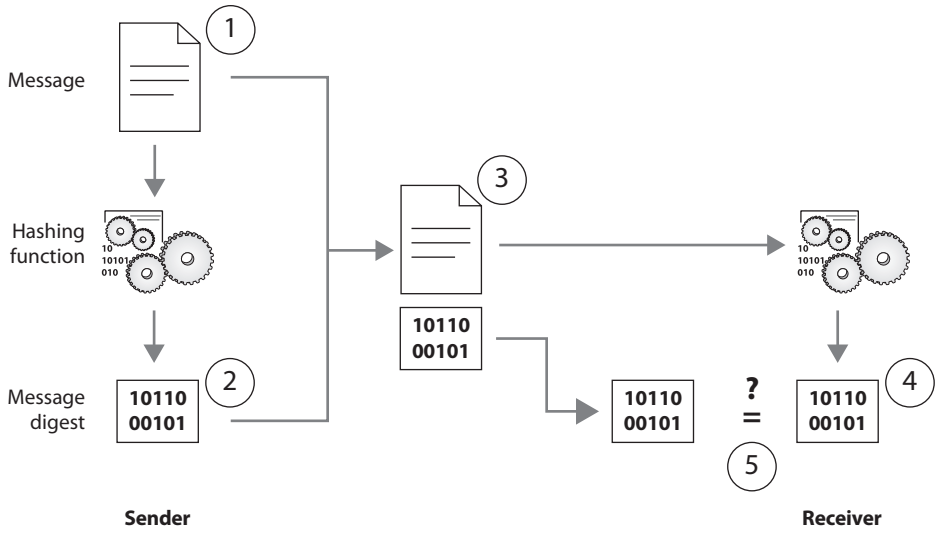


Figure 8-16 Verifying message integrity with a message digest

Message Authentication Code

If Cheryl wanted more protection than just described, she would need to use a *message authentication code (MAC)*, an authentication scheme derived by applying a secret key to a message in some form. This does not mean the symmetric key is used to encrypt the message, though. A good example of a MAC leverages hashing functions and is called a *hash MAC (HMAC)*.

In the previous example, if Cheryl were to use an HMAC function instead of just a plain hashing algorithm, a symmetric key would be concatenated with her message. The result of this process would be put through a hashing algorithm, and the result would be a MAC value. This MAC value would then be appended to her message and sent to Scott. If Bruce were to intercept this message and modify it, he would not have the necessary symmetric key to create the MAC value that Scott will attempt to generate. Figure 8-17 shows the following steps to use an HMAC:

1. The sender writes a message.
2. The sender concatenates a shared secret key with the message and puts them through a hashing function, generating a MAC.
3. The sender appends the MAC value to the message and sends it to the receiver. (Just the message with the attached MAC value. The sender does not send the symmetric key with the message.)
4. The receiver concatenates his copy of the shared secret key with the message and puts the results through a hashing algorithm to generate his own MAC.
5. The receiver compares the two MAC values. If they are the same, the message has not been modified.

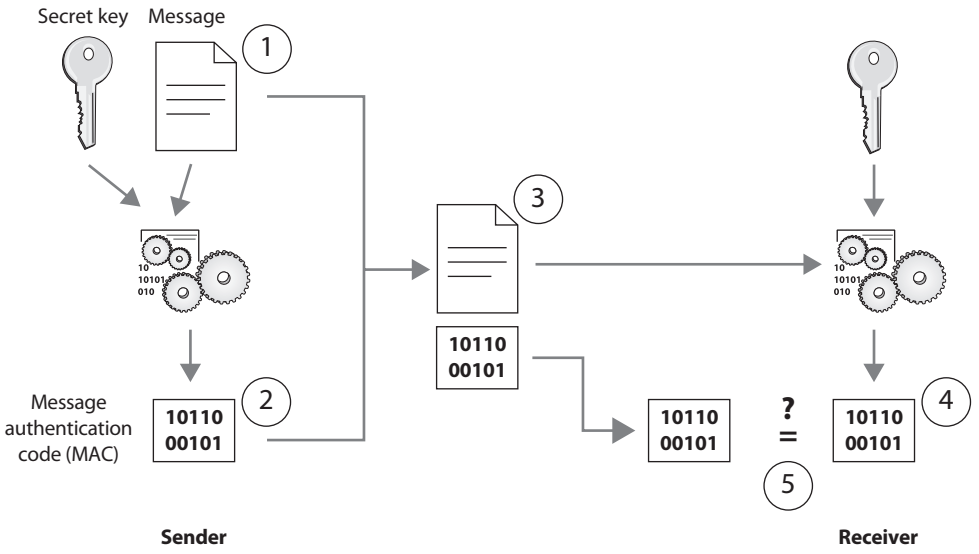


Figure 8-17 Verifying message integrity with a message digest

Now, when we say that the message is concatenated with a symmetric key, we don't mean a symmetric key is used to encrypt the message. The message is not encrypted in an HMAC function, so there is no confidentiality being provided. Think about throwing a message in a bowl and then throwing a symmetric key in the same bowl. If you dump the contents of the bowl into a hashing algorithm, the result will be a MAC value.

Digital Signatures

A MAC can ensure that a message has not been altered, but it cannot ensure that it comes from the entity that claims to be its source. This is because MACs use symmetric keys, which are shared. If there was an insider threat who had access to this shared key, that person could modify messages in a way that could not be easily detected. If we wanted to protect against this threat, we would want to ensure the integrity verification mechanism is tied to a specific individual, which is where public key encryption comes in handy.

A *digital signature* is a hash value that has been encrypted with the sender's private key. Since this hash can be decrypted by anyone who has the corresponding public key, it verifies that the message comes from the claimed sender and that it hasn't been altered. The act of signing means encrypting the message's hash value with a private key, as shown in Figure 8-18.

Continuing our example from the previous section, if Cheryl wants to ensure that the message she sends to Scott is not modified *and* she wants him to be sure it came only from her, she can digitally sign the message. This means that a one-way hashing function would be run on the message, and then Cheryl would encrypt that hash value with her private key.

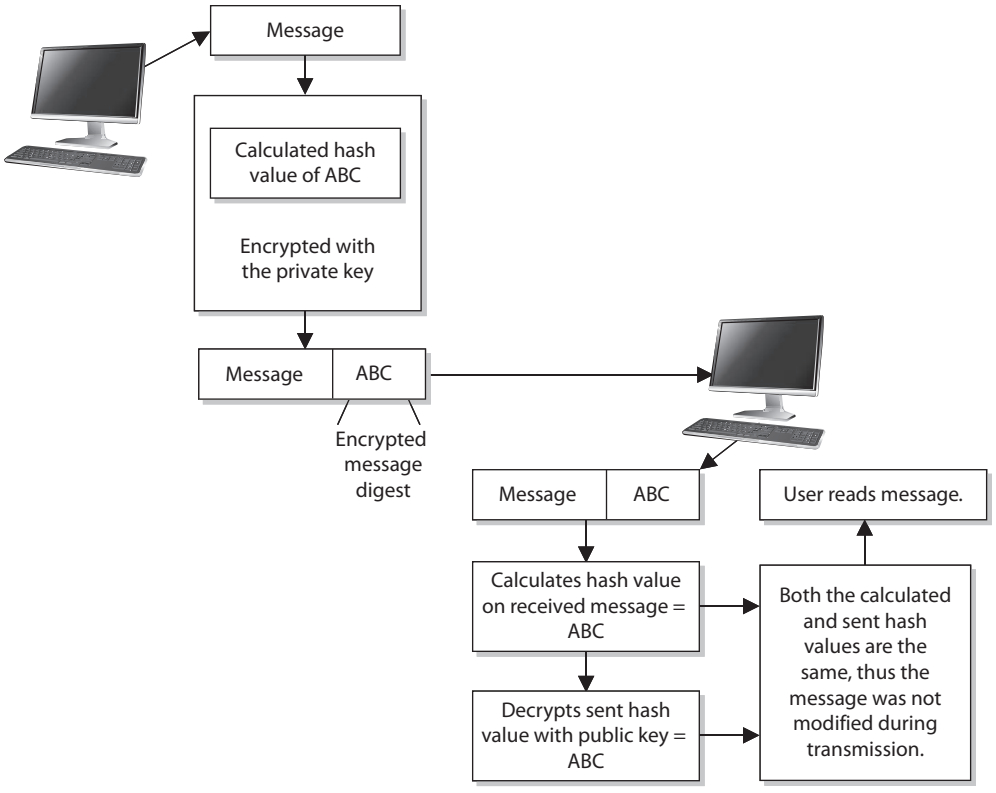


Figure 8-18 Creating a digital signature for a message

When Scott receives the message, he performs the hashing function on the message and comes up with his own hash value. Then he decrypts the sent hash value (digital signature) with Cheryl's public key. He then compares the two values, and if they are the same, he can be sure the message was not altered during transmission. He is also sure the message came from Cheryl because the value was encrypted with her private key.

The hashing function ensures the integrity of the message, and the signing of the hash value provides authentication and nonrepudiation. The act of signing just means the value was encrypted with a private key.

Because digital signatures are so important in proving who sent which messages, the U.S. federal government decided to establish standards pertaining to their functions and acceptable use. In 1991, NIST proposed a federal standard called the *Digital Signature Standard (DSS)*. It was developed for federal departments and agencies, but most vendors also designed their products to meet these specifications. The federal government requires its departments to use DSA, RSA, or the elliptic curve digital signature algorithm (ECDSA) and SHA-256. SHA creates a 256-bit message digest output, which is then inputted into one of the three mentioned digital signature algorithms. SHA is used to ensure the integrity of the message, and the other algorithms are used to digitally sign

the message. This is an example of how two different algorithms are combined to provide the right combination of security services.

RSA and DSA are the best-known and most widely used digital signature algorithms. DSA was developed by the NSA. Unlike RSA, DSA can be used only for digital signatures, and DSA is slower than RSA in signature verification. RSA can be used for digital signatures, encryption, and secure distribution of symmetric keys.

Digests, HMACs, and Digital Signatures—Oh My!

MACs and hashing processes can be confusing. The following table simplifies the differences between them.

Function	Steps	Security Service Provided
Hash	1. Sender puts a message through a hashing algorithm and generates a message digest (MD) value. 2. Sender sends message and MD value to receiver. 3. Receiver runs just the message through the same hashing algorithm and creates an independent MD value. 4. Receiver compares both MD values. If they are the same, the message was not modified.	Integrity; not confidentiality or authentication. Can detect only unintentional modifications.
HMAC	1. Sender concatenates a message and secret key and puts the result through a hashing algorithm. This creates a MAC value. 2. Sender appends the MAC value to the message and sends it to the receiver. 3. The receiver takes just the message and concatenates it with her own symmetric key. This results in an independent MAC value. 4. The receiver compares the two MAC values. If they are identical, the receiver knows the message was not modified.	Integrity and data origin authentication; confidentiality is not provided.
Digital signature	1. The sender computes the hash of the message and encrypts it with her private key. 2. The sender appends the encrypted message digest to the message and sends it to the receiver. 3. The receiver computes the hash of the received message. 4. The receiver decrypts the received message digest using the sender's public key. 5. The receiver compares the two digests. If they are identical, the receiver knows the message was not modified and knows from which system it came.	Integrity, sender authentication, and nonrepudiation; confidentiality is not provided.

Public Key Infrastructure

Now that you understand the main approaches to modern cryptography, let's see how they come together to provide an infrastructure that can help us protect our organizations in practical ways. A *public key infrastructure (PKI)* consists of programs, data formats, procedures, communication protocols, security policies, and cryptosystems working in a comprehensive manner to enable a wide range of dispersed people to communicate in a secure and predictable fashion. In other words, a PKI establishes and maintains a high level of trust within an environment. It can provide confidentiality, integrity, nonrepudiation, authentication, and even authorization. As we will see shortly, it is a *hybrid* system of symmetric and asymmetric cryptosystems, which were discussed in earlier sections.

There is a difference between public key cryptography and PKI. Public key cryptography is another name for asymmetric algorithms, while PKI (as the name states) is an infrastructure that is partly built on public key cryptography. The central concept in PKI is the digital certificate, but it also requires certificate authorities, registration authorities, and effective key management.

Digital Certificates

Recall that, in asymmetric key cryptography, we keep a private key secret and widely share its corresponding public key. This allows anyone to send us an encrypted message that only we (or whoever is holding the private key) can decrypt. Now, suppose you receive a message from your boss asking you to send some sensitive information encrypted with her public key, which she attaches to the message. How can you be sure it really is her? After all, anybody could generate a key pair and send you a public key claiming to be hers.

A *digital certificate* is the mechanism used to associate a public key with a collection of components in a manner that is sufficient to uniquely identify the claimed owner. The most commonly used standard for digital certificates is the International Telecommunications Union's *X.509*, which dictates the different fields used in the certificate and the valid values that can populate those fields. The certificate includes the serial number, version number, identity information, algorithm information, lifetime dates, and the signature of the issuing authority, as shown in Figure 8-19.

Note that the certificate specifies the subject, which is the owner of the certificate and holder of the corresponding private key, as well as an issuer, which is the entity that is certifying that the subject is who they claim to be. The issuer attaches a digital signature to the certificate to prove that it was issued by that entity and hasn't been altered by others. There is nothing keeping anyone from issuing a self-signed certificate, in which the subject and issuer can be one and the same. While this might be allowed, it should be very suspicious when dealing with external entities. For example, if your bank presents to you a self-signed certificate, you should not trust it at all. Instead, we need a reputable third party to verify subjects' identities and issue their certificates.