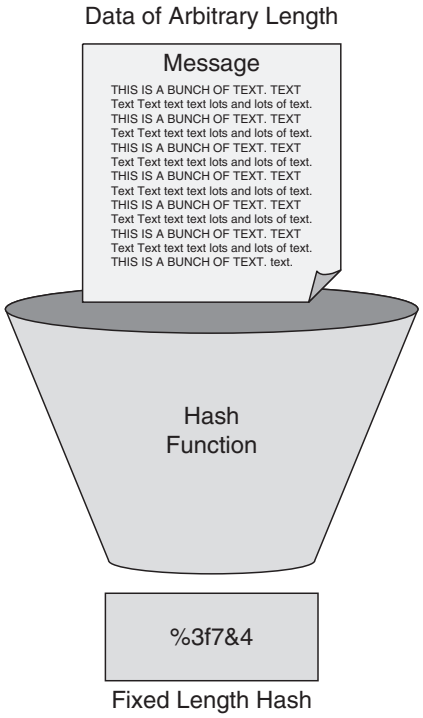
Understanding Cryptographic Hashes

Hashing is used to provide data integrity. Hashes are based on one-way mathematical functions. These can be easy to compute but extremely challenging to reverse. The process of hashing and the difficulty of reversing the hash is akin to scrambling an egg and then trying to put it back together again.

Working with Hashing

The way that hashing works in practice is that data of arbitrary length is input into the hash function and then is processed through the function, resulting in a fixed-length hash. The resultant fixed-length hash is called a digest or fingerprint. Figure 12-7 shows the hashing process.

Figure 12-7 *Hashing Process*



If you are familiar with the calculation of cyclic redundancy check (CRC) checksums, hashes are quite similar to this but are cryptographically much stronger. If you're not too familiar with CRC, if you have the CRC value, it is relatively easy to generate data with the same CRC. Because of the strength of hash functions, it is computationally infeasible for an attacker to possess two separate sets of data that would come up with the same fingerprint.

Designing Key Management

One of the most challenging aspects of designing a cryptosystem is planning for key management. In fact, cryptosystem failures have occurred because of shortcomings in key management. Each of the current cryptographic algorithms requires the services of key management procedures, making this an extremely important area to consider. For an attacker, the general target when seeking to attack a cryptographic system is the key management system, rather than the algorithm itself.

Components of Key Management

When considering key management, you must consider several components that address the life cycle of key management from generation to destruction:

- Key generation
- Key verification
- Key storage
- Key exchange
- Key revocation and destruction

Modern cryptographic systems generate keys automatically, rather than leaving it to the end user. To help ensure that all keys are likely to be equally generated, so that the attacker cannot predict which keys are likely to be used, quality random-number generators are necessary. It is not uncommon for a cryptographic algorithm to have some weak keys that should not be used. Proper key verification procedures should be used to regenerate these keys when they occur.

Key storage is another factor that must be considered. With today's modern multiuser operating systems that work with cryptography, a key can be stored in memory. When memory is swapped to the disk, it presents a possible problem, because a Trojan horse program, if installed on the PC, could then gain access to that user's private keys.

A secure key exchange mechanism is also necessary. It should allow secure agreement on the keying material with the other party, likely over an untrusted medium.

The final element of good key management to consider is key revocation and destruction. The process of key revocation notifies all the parties involved that a given key has been compromised and should not be used. Key destruction goes beyond this by erasing old keys so that a malicious attacker cannot recover them.

Understanding Keyspaces

The keyspace of an algorithm represents a defined set of all possible key values. For each key of n bits, a keyspace is produced that has 2^n possible key values. This means that adding 1 bit to the key would effectively double the size of the keyspace.

Let's look at DES by way of an example. DES employs a 56-bit key and with this produces a keyspace of more than 72,000,000,000,000,000 (2^{56}) potential keys. If we were to add 1 bit to the key length, the keyspace would double. That means that an attacker would need twice as much time to search the keyspace.

No algorithm is without some weak keys in its keyspace, as discussed previously. These weak keys may enable an attacker to break the encryption via a shortcut. So what exactly constitutes a weak key?

A key is said to be weak when it shows regularities in encryption or poor encryption. A good example is DES. DES has four keys for which encryption is exactly the same as decryption. Should one of these weak keys be encrypted twice, the original plain text would be recovered.

The chance that such keys would be chosen is almost unimaginable. However, each implementation should still verify all keys and take steps to prevent weak keys from being used. This is particularly the case with manual key generation, so you should take special care to avoid defining weak keys.

Issues Related to Key Length

As we have discussed, the only way to break a proven cryptographic system is with a brute-force attack. These attacks search the entire keyspace, trying all possible keys, until the key that decrypts the data is found. To defend against this form of attack, the keyspace must be sufficiently large enough that such a search would require an enormous amount of time, rendering this form of attack impractical.

Even for successful brute-force attacks, generally an attacker has to search half the keyspace to find the correct key. Of course, the time required for such a search depends on the computer resources available to the attacker. With key lengths of significant size, such an attack could take many millions, if not billions, of years to yield success.

The protection strength of modern trusted algorithms depends exclusively on the length of the key. You should select a key length so that it protects data confidentiality or integrity for an adequate period of time. The more sensitive the data, and the longer the period required for secrecy, the longer the key that must be used.

When considering the level of protection required, you must also take into account the characteristics of those likely to attack your data. For instance, you must estimate the attacker's resources and how long you must protect the data.

For example, suppose a would-be attacker has $1 million of funding that can be used toward the attack, and the data must be protected for a period of no less than one year. What form

of encryption should we choose? Classic DES would not be a good choice, because it can be broken by a $1 million machine in only a couple of minutes. If instead we employed 168-bit 3DES or even 128-bit RC4, it would take that same attacker, funded with that same $1 million, a million years or more to crack into your data. Considering our attacker, his funding and our need for security are both important in selecting the proper key length.

Another issue impacting the choice of key length is performance. It is important to strive to balance speed and protection strength for the selected algorithm. Certain algorithms, such as RSA, take much longer to run with large key sizes. We should strive for adequate protection, without hindering communication over untrusted networks.

Finally, you also need to be aware that because of rapid advances in technology and cryptanalytic methods, what may be an adequate key size today may quickly no longer be appropriate. The National Institute of Standards and Technology (NIST) offers recommendations on adequate key lengths for various applications. You may review these on the NIST website at http://www.keylength.com/en/4/.

SSL VPNs

Security on the Internet for such applications as web browsing, e-mail, Internet faxing, instant messaging, and other forms of data transfer is provided by cryptographic protocols. Among the most popular choices to support these applications are Transport Layer Security (TLS) and its predecessor, Secure Socket Layer (SSL). Although there are subtle differences between SSL and TLS, the actual protocol remains quite similar.

Both SSL and TLS support a variety of cryptographic algorithms, or ciphers, to help provide such functions as authenticating the server and client to each other, transmitting certificates, and establishing session keys. For bulk encryption, symmetric algorithms are used. Asymmetric algorithms are used for authentication and key exchange. Hashing is used as part of the authentication process.

With an SSL-based VPN, you can easily provide remote-access connectivity from almost any Internet-enabled location. All that is needed is a standard web browser and its native SSL encryption. There is no need for special-purpose client software on the remote system. This flexibility allows SSL VPNs to provide “anywhere” connectivity from corporate desktops, as well as from noncompany-managed desktops. Employees may use the SSL-based VPN to connect from home on their own PCs, contractor or business partner desktops can also be easily connected, and users can even connect via Internet kiosks. Through dynamic download, clients are supplied with all the software needed for application access across the SSL VPN connection. This feature dramatically minimizes the maintenance of desktop software.

If you need to support the remote resource needs of a diverse user base, SSL VPNs and IPsec VPNs provide complementary technologies that can be deployed together to meet these needs. Each of these VPN solutions offers access to virtually any network application or resource from a remote location. However, SSL VPNs do offer some additional features, allowing for easy connectivity from desktops outside your company's management, as well as little or no desktop software maintenance, and user-customized web portals upon login.

Establishing an SSL Tunnel

Let's examine the steps involved in establishing an SSL tunnel (see Figure 12-8):

Step 1 The user makes an outbound connection to TCP port 443.

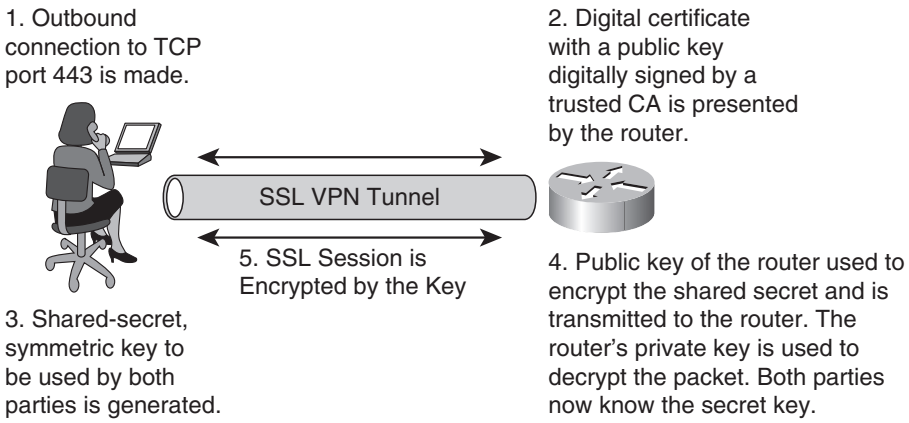
Step 2 The router presents a digital certificate that contains a public key that is digitally signed by a trusted certificate authority (CA).

Step 3 The user's computer generates a shared-secret symmetric key that will be used by both parties.

Step 4 The router's public key is used to encrypt the shared secret and is transmitted to the router. The router's software uses the private key to decrypt the packet. When this is complete, both parties in the session know the secret key.

Step 5 The key encrypts the SSL session.

Figure 12-8 *Establishing an SSL Tunnel*



With the SSL tunnel established, two parties may securely transmit data. By using the mechanisms discussed here, you can effectively extend the reach of your corporate network, allowing users to easily and securely gain access to corporate resources from wherever they may be.

Foundation Topics

Examining Hash Algorithms

For centuries everyone from kings to generals to college students has wanted to ensure the authenticity of their communications. In this section we will examine the role of hash algorithms in helping provide this assurance through the process of *hashing*. Along the way we will examine hash functions and learn about HMAC, as well as explore MD5 and SHA-1. Let's begin with a brief overview of what a hash function does.

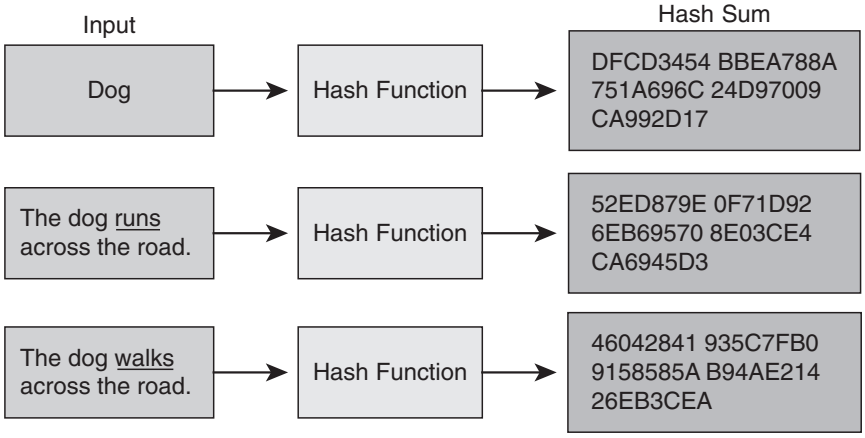
A hash function is a means of turning data into a relatively small number that then may act as a digital fingerprint of the data. The algorithm that is used substitutes or transposes the data to create this unique fingerprint. These fingerprints may be called hash sums, hash values, hash codes, or just hashes. Cryptographic hashes serve a variety of purposes in information security applications. They are used to do message integrity checks and provide digital signatures in various information security applications, such as authentication and message integrity.

Exploring Hash Algorithms and HMACs

Figure 13-1 is an example of how a simple sentence can be transformed using a hash function to yield a cryptographic result. You can see that changing a single word alters the hash output.

Key Topic

Figure 13-1 *Hashing Example*



Changing a single word in the text alters the output of the hash function.

Although you might not need to hash a simple sentence like this, many other applications exist in terms of network security. Hashes can be employed to help secure data as it traverses your network, as well as to secure login authentication credentials as a user is validated before accessing network resources.

Anatomy of a Hash Function

A variety of hash functions exist, but they all share the common characteristic that they are built for speed and are designed to yield very few hash collisions in their expected input domains. A hash "collision" (sometimes called a "hash clash") happens when two distinct inputs entered into a hash function produce identical outputs. Each hash function has the potential for collisions, but if you are working with a well-designed hash function, collisions should occur less frequently. In terms of hash functions, collisions inhibit the distinguishing of data, making records more costly to find in hash tables and data processing.

Another characteristic of hash functions is that they must be deterministic. In other words, if a hash function generates two hashes that are different, we can conclude that the two inputs were different in some way.

Hash values that are computed may be the same for different input values. This may seem odd, but it is because of the general requirement that the hash value must be able to be stored in fewer bits than the data that is being hashed. A primary design goal of hash functions is to minimize the likelihood of a hash collision.

One desirable property of a hash function is the mixing property. What this means is that a small change in the input (1 bit) should cause a large change in the output (about half of the bits). This significant change in the outcome is called the avalanche effect.

Most hash functions have what is called an infinite domain. This might be something like byte strings of arbitrary length. They also have a finite range—for instance, bit sequences of a fixed length. In some applications, hash functions may be designed with a one-to-one mapping between an identically sized domain and range. Hash functions such as these, which are one-to-one, are also called permutations. For these hash functions, reversibility is achieved by using a series of reversible "mixing" operations on the function input.

Application of Hash Functions

Hash functions may be used for a variety of applications; therefore, they are often tailored to a given need. Cryptographic hash functions begin with the assumption that an adversary can deliberately try to find inputs with the same hash value. The creation of a well-designed cryptographic hash involves a one-way operation in which no practical way exists to calculate a particular data input that will result in a desired hash value. This one-way nature

makes the hash very difficult to forge. Message Digest 5 (MD5) is an example of a function intended for cryptographic hashing that is commonly used as a stock hash function.

When a function is used for error detection and correction, it focuses on distinguishing those cases in which data has been disturbed by a random process. One way that a hash function might be employed is as a checksum. Hash functions have a relatively small hash value that can be used to verify that a data file of any size has not been altered, making them valuable in network security implementations.

In terms of real-world application, perhaps an example is in order. One of the most common applications of hash functions is helping prove the authenticity of a message. When a hash function is employed in this manner, it produces a "fingerprint" of the message. This fingerprint is the hash code or message digest, which is the output of the hash function. This is used to create a digital signature for the message to ensure its authenticity. We will explore the concept of digital signatures as well as various hash functions in later sections.

Cryptographic Hash Functions

Put simply, a cryptographic hash function takes an input and returns a fixed-length string, which is called the hash value or hash sum. These hash functions, as mentioned in the preceding section, may be used for a variety of purposes, including cryptography. A hash value, as complex as it may become, is, on the surface, simply a concise representation of a longer message or document from which it was derived. The output of the hash function, often called the message digest, is a sort of "digital fingerprint" of the larger document. Message integrity checks and digital signatures, used for various security applications such as authentication and message integrity, can be provided by cryptographic hash functions.

The way this works is that the hash function accepts a string of variable length, sometimes called a message, as an input and then produces a fixed-length string, called a message digest or digital fingerprint, as an output. A hash value, often called a "digest" or "checksum," is a type of signature that represents the contents of a stream of data.

Put another way, a hash is akin to the seal on certain over-the-counter medications. A plastic film covers the top of the bottle. If it has been disturbed, it is evident that that the medication has been tampered with. The hash serves this same function, only as a digital mechanism.

Two of the most widely used hash functions are MD5 and SHA-1 (Secure Hash Algorithm 1). However, in 2005 security flaws were identified in both of these common algorithms. Although these hash functions are indeed flawed, they are still widely used today. Why is this?

As cryptographic analysts seek to break hash functions, sometimes they are successful. However, this is often after many years and with the aid of computing technology outside

of what would be considered the norm. So even though these two hash functions have security flaws, it's very unlikely that an attacker, under normal and reasonable circumstances, could exploit them. Therefore, these two functions are still widely used for a variety of purposes today. We will explore both MD5 and SHA-1 and their application to networking technologies later in this chapter.

When we consider the security of a cryptographic hash function, it should behave as much as possible like a random function while still being efficiently computable and deterministic in nature.

If either of the following statements is computationally feasible, a cryptographic hash function is considered insecure:

- A previously unseen message that matches a given digest
- Two different messages having the same message digest, called a collision

If an attacker can discover either of these things, he may be able to exploit the vulnerability in the hash function. For instance, he might be able to substitute an unauthorized message for one that has been authorized.

With a truly secure cryptographic hash function, it should not be possible to find two messages whose message digests are substantially similar, let alone exactly the same. Likewise, with a secure cryptographic hash, an attacker should not be able to learn anything of value about a message from only its digest. However, if an attacker can compromise security and obtain the digest, this could prove valuable should that same message occur again.

Application of Cryptographic Hashes

Let's examine a cryptographic hash to better understand how it works.

Suppose Anthony presents Tom with a rather difficult math problem that he claims to have solved. Tom wants to try to solve the problem himself, but he also wants to be sure that Anthony is telling the truth about having solved it. Anthony writes down his solution and then appends a random nonce, computes its hash, and tells Tom this hash value. The nonce that Anthony uses in this case is a random or pseudorandom number that is used only once for the purposes of this communication. All Tom has is the hash value; Anthony keeps the solution and the nonce secret. Tom gets to work on the math problem. When Tom finally solves the problem, Anthony can prove that he solved the problem as well by telling Tom the nonce. This example employs what cryptographers call a simple commitment scheme. In the world of computer networking, "Tom" and "Anthony" might very well be two computer programs attempting to prove a message's authenticity.

Secure hashes often provide a mechanism to verify message integrity. Let's say that a file is to be sent across the Internet, and you are concerned that it might be intercepted and altered in transit. Using a secure cryptographic hash would allow you to determine whether any changes have been made to the file. For example, you could do this by comparing message digests calculated before, and after, transmission of the file over the public network.

One means to reliably identify a file is a message digest. For instance, the Git source code management system uses the sha1sum program to examine various types of content such as file content, directory trees, ancestry information, and the like to uniquely identify a file.

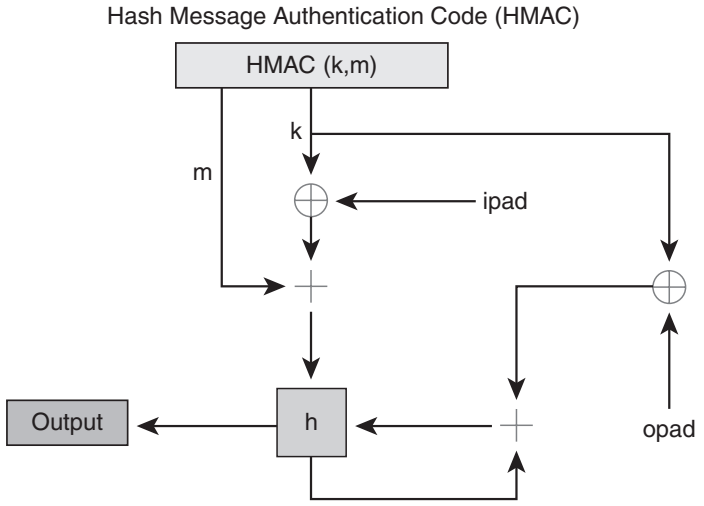
Another frequent use of cryptographic hashes is password verification. When we think in terms of passwords, we know that protecting them is of the utmost importance. That is why passwords generally are not stored in clear text but rather in a digest form. When a digest is used, a user is authenticated in the following manner: The user provides her username and password. The password she presents is hashed and then compared to the hash that has been stored. If a match is found, the user is granted access. This type of cryptographic hash generally is called one-way encryption.

SHA-1, MD5, and as RIPEMD-160 are some of the most widely used message digest algorithms. This is true even though in August 2004, researchers found weaknesses in a number of hash functions, including MD5, SHA-0, and RIPEMD. These findings also have called into question the long-term security of later algorithms derived from these hash functions—in particular, SHA-1, which is a strengthened version of SHA-0. As recently as August 2005, an attack against SHA-1 found collisions in 2^{63} operations. An attack in February of that same year found collisions in about 2^{69} hashing operations, rather than the 2^{80} expected for a 160-bit hash function. Findings such as these call into question the security of these common cryptographic hashes. However, in terms of practical applications, these collisions do not warrant stopping the use of these extremely popular hash functions.

HMAC Explained

Keyed Hash-based Message Authentication Code (HMAC) in cryptographic terms is a type of message authentication code (MAC) calculated by using a cryptographic hash function along with a secret key. It may be used to simultaneously verify the data's integrity and the message's authenticity. An iterative cryptographic hash function such as MD5 or SHA-1 may be used to calculate the HMAC. When these are used, the resulting MAC algorithm is called HMAC-MD5 or HMAC-SHA-1, for instance. The cryptographic strength of the underlying hash function, along with the size and quality of the key and the size of the hash output length in bits, define the cryptographic strength of the HMAC. Figure 13-2 illustrates HMAC.

Figure 13-2 *HMAC*



h = Cryptographic hash function.
m = Message to be authenticated.
k = Secret key padded with extra 0's (ipad/opad) to the block size of the hash function.
ipad = Inner padding.
opad = Outer padding.

Iterative hash functions, such as MD5 and SHA-1, break a message into blocks of a fixed size and then iterate over them with a compression function. For instance, MD5 and SHA-1 operate on 512-bit blocks. As mentioned, the size of the HMAC output is the same as that of the underlying hash function (128 or 160 bits in the case of MD5 and SHA-1), but you can truncate this if you want to. When the hash image is truncated, the security of the MAC is reduced.

In 1996 Mihir Bellare, Ran Canetti, and Hugo Krawczyk wrote about the construction and analysis of HMACs. These authors also wrote RFC 2104 in 1997 and FIPS PUB 198, which generalizes and standardizes the use of HMACs. Both the IPsec and TLS protocols use HMAC-SHA-1 and HMAC-MD5.

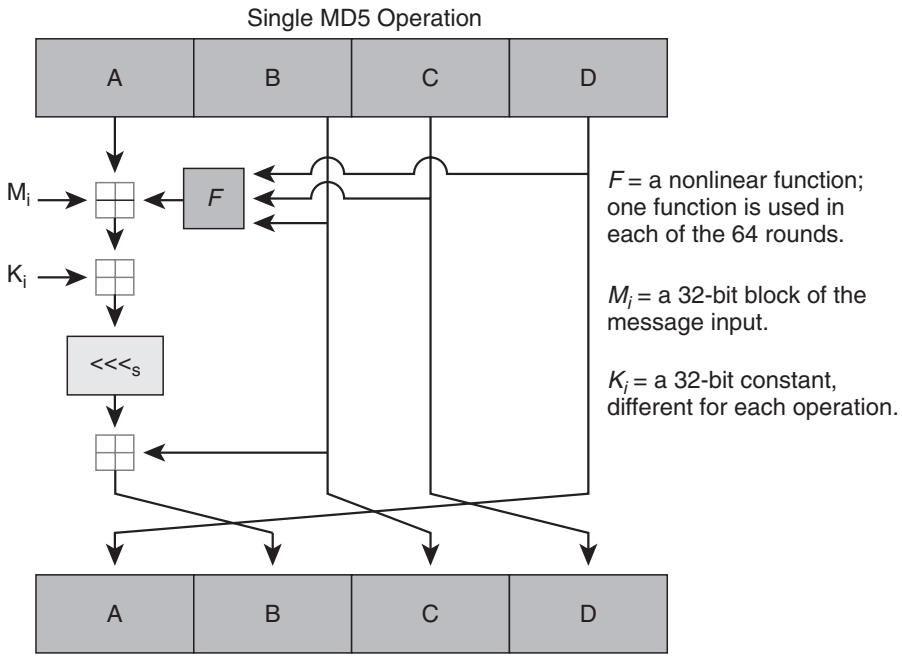
MD5 Features and Functionality

Defined in RFC 1321, MD5 (Message Digest algorithm 5), with its 128-bit hash value, has been employed in a wide variety of security applications. It is also commonly used to check the integrity of files. An MD5 hash typically is expressed as a 32-character hexadecimal number.

Figure 13-3 shows a single MD5 operation. In practice, MD5 consists of 64 of these operations. These are grouped in four rounds of 16 operations. In this figure, F is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, which is different for each operation.

Key Topic

Figure 13-3 *MD5 Algorithm*



Ronald Rivest designed MD5 in 1991 as a replacement for the earlier MD4 hash function. Five years later, in 1996, a flaw was found in the design of MD5. Although this flaw was not a fatal weakness, the cryptography community began recommending the use of other algorithms, such as SHA-1. Ironically, the widely used SHA-1 algorithm has since been shown to be vulnerable as well, as noted earlier. In 2004, researchers discovered more serious flaws in the algorithm, calling into question the use of the algorithm for certain security purposes.

Origins of MD5

Ronald Rivest of MIT created Message Digest as a series of message digest algorithms. MD5 was designed to be a secure replacement for its predecessor, MD4, when work demonstrated that MD4 was likely unsecure. Security of the MD5 algorithm was initially brought into question in 1993 when researchers found a pseudo-collision of the MD5 compression function. In other words, two different initialization vectors produced an identical digest.

In 1996, a true collision of the MD5 compression function was announced. Although this was not an attack on the full MD5 hash function, it was close enough for cryptographers to recommend switching to a replacement. Among the recommendations were WHIRLPOOL, SHA-1, and RIPEMD-160.

The hash, at only 128 bits, was small enough for cryptographers to fear that it would be vulnerable to a birthday attack. A birthday attack is a type of cryptographic attack that takes advantage of the mathematics behind the birthday paradox. The birthday paradox addresses

the probability that, in a set of randomly chosen people, two of them will have the same birthday. If the random group consists of 23 or more people, the probability that two of them will have the same birthday is greater than 50%. If the number of people in the pool grows to 57, the probability of a shared birthday is more than 99%. The probability approaches 100% as the number of individuals grows.

To test this theory, MD5CRK, a distributed project, was started in March 2004. The aim of the project was to demonstrate that MD5 is practically insecure by finding a collision using a birthday attack.

In less than six months, MD5CRK ended in August 2004 when collisions for the full MD5 were announced. The reported attack took only one hour on an IBM p690 cluster.

Further weaknesses were discovered in March 2005. A team of researchers constructed two x.509 certificates with different public keys and the same MD5 hash, a demonstrably practical collision. Within days, additional researchers announced an improved algorithm that could construct MD5 collisions in a few hours on a single notebook computer. Further hardship followed when, on March 18, 2006, an algorithm was published that could find a collision within one minute on a notebook computer using a method called tunneling.

Vulnerabilities of MD5

MD5 makes only a single pass over data. Because of this, if two prefixes with the same hash can be constructed, it is possible to add a common suffix to both to make the collision reasonably more possible.

Currently there exist collision-finding techniques that allow the preceding hash state to be specified arbitrarily. Therefore, a collision can be found for any desired prefix. This means that for any given string of characters X (for instance, a password), two colliding files can be determined that both begin with X.

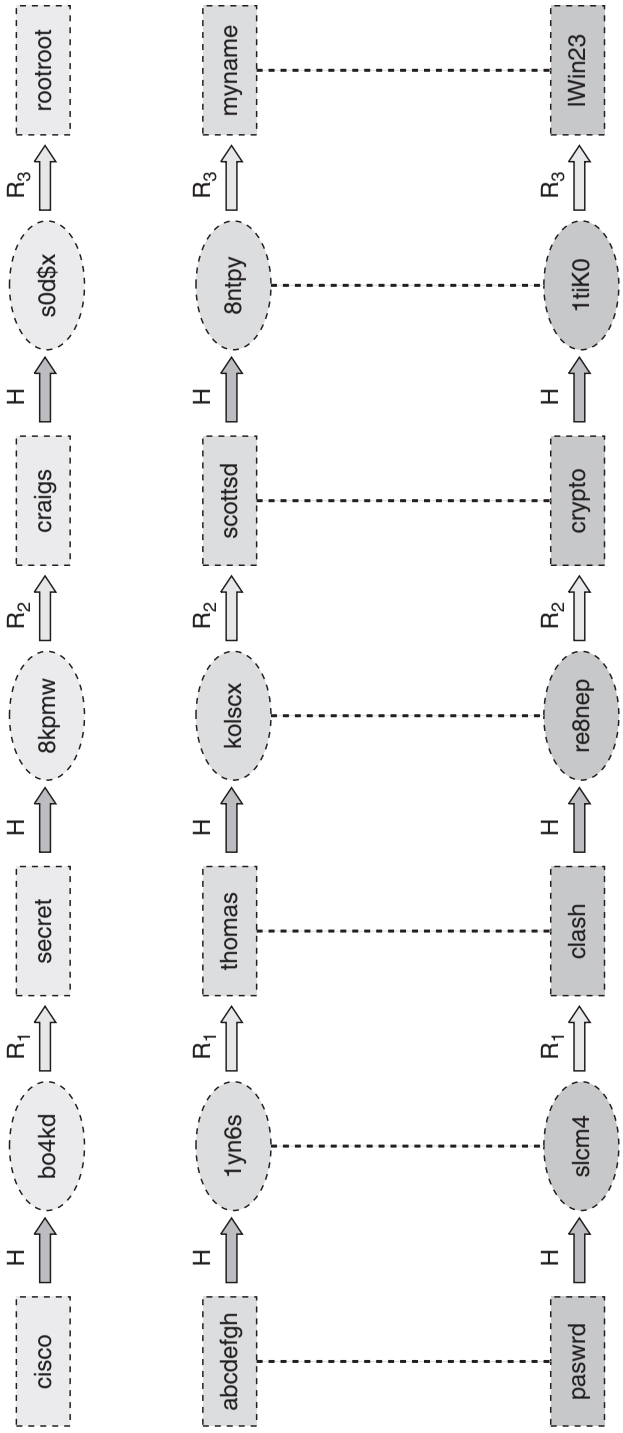
To generate these two colliding files, all that is needed is a template file, with a 128-byte block of data aligned on a 64-byte boundary, that can be changed freely by the collision-finding algorithm.

Figure 13-4 shows a rainbow table. Attackers can use rainbow tables to try to reverse hashes into strings.

Attackers can use MD5 rainbow tables, which are easily accessible online, to reverse many MD5 hashes into strings that collide with the original input. The general purpose of such attacks is password cracking. One means of defense is to combine passwords with a salt (a series of random bits added to the password) before the MD5 digest is generated. This combination makes rainbow tables much less useful.

Key Topic

Figure 13-4 *Rainbow Table*



Usage of MD5

One of the most common uses of MD5 digests is to provide some degree of assurance that a transferred file has arrived intact. An example of this is when a file server provides a precomputed MD5 checksum for a file. To ensure that the file has not been tampered with, the user can compare the checksum of the downloaded file to the one provided by the file server. Various UNIX-based operating systems include MD5 checksum utilities. For those working with Windows operating systems, this capability generally is provided by third-party applications.

Recent vulnerabilities that have been discovered with MD5 now make it easy to generate MD5 collisions. That being the case, it is possible for the person who created the file to create a second file with the same checksum, negating the protection against some forms of malicious tampering. At other times, the checksum might not be trustworthy, such as if it was obtained over the same channel as the downloaded file. In a case such as this, MD5 can only provide error checking, allowing it to recognize a corrupt or incomplete download. This is often the case when you download large files.

Another common usage of MD5 is to store passwords. To provide a defense against the vulnerabilities we have discussed, a salt can be added to the password before it is hashed. In fact, some implementations of this technique apply the hashing function multiple times to provide greater security.

SHA-1 Features and Functionality

SHA-1 (Secure Hash Algorithm 1) is one of five cryptographic hash functions referred to as SHA hash functions. They were designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard. Much like the MD5 hash algorithm, SHA-1 computes a fixed-length digital representation (a *message digest*) from an input data sequence (the *message*) of any length.

A hash such as this is defined as "secure" when it is computationally infeasible to

- Find a message that corresponds to a given message digest
- Find two different messages that produce the same message digest

When working with a secure hash such as this, any change to a message should, with a very high degree of probability, result in a different message digest.

Figure 13-5 shows the SHA-1 hash and how it is used. We will discuss the application of SHA-1 in greater detail in an upcoming section.

Figure 13-5 *SHA-1 Hash*

```
SHA1("The quick brown fox jumps over the lazy dog")
= 2fd4e1c6 7a2d28fc ed849ee1 bb76e739 1b93eb12
```

With SHA-1, a small change in the message will result in a completely different hash due to the avalanche effect. For example, changing *dog* to *log*:

```
SHA1("The quick brown fox jumps over the lazy log")
  = de9f2c7f d25e1b3a fad3e85a 0bd17d9b 100db4b3
```

Overview of SHA-1

The SHA-1 hash produces a message digest that is 160 bits long, as opposed to 128 bits for MD5. A number of widely used security applications and protocols employ SHA-1, including TLS, SSL, PGP, SSH, S/MIME, and IPsec. SHA-1 has been positioned as the successor to MD5, which was one of the most widely used hash functions until SHA-1 was introduced.

Like its predecessor, MD5, researchers have attempted to validate the security of SHA-1. Although it has been somewhat compromised, no attacks have yet been reported on the SHA-2 variants, which are algorithmically similar to SHA-1.

To expand on these SHA-2 variants, NIST has published four additional hash functions in the SHA family. Each has a longer digest. Collectively they are known as SHA-2. Each of these individual variants is named after its digest length (in bits): SHA-224, SHA-256, SHA-384, and SHA-512. The final three were first published in 2001 in the draft Federal Information Processing Standard publication (FIPS PUB) 180-2. This publication (FIPS PUB 180-2) also includes SHA-1 and was released as an official standard in 2002. A change notice was published for FIPS PUB 180-2 in February 2004. This specified an additional variant, SHA-224. These variants have not yet received the degree of scrutiny from the cryptographic community that SHA-1 has, so their cryptographic security is not yet as well-established.

Currently efforts are under way within the cryptography community to develop improved alternative hashing algorithms. In one such effort, NIST is seeking to develop one or more additional hash algorithms through a public competition, similar to the development process used for the Advanced Encryption Standard (AES). It is the hope of NIST that a new standard will be developed and announced in 2012.

Vulnerabilities of SHA-1

Some researchers working with SHA-1 have called into question its use in new cryptosystems. NIST has announced that it plans to phase out the use of SHA-1 by 2010 in favor of the SHA-2 variants. To better understand the vulnerabilities of SHA-1 and why NIST is searching for a replacement, you need to better understand the research that has been done into its weaknesses.

SHA-1 has fallen victim to various attacks that call into question its cryptographic strength. One such attack, published by researchers in early 2005, found collisions when working with a reduced version of SHA-1. In February of that same year, a team of researchers announced an attack that could find collisions in the full version of SHA-1, requiring fewer than 2^{69} operations. To put this in perspective, a brute-force search attack would require 2^{80} operations.

When asked how these attacks were possible, researchers pointed to the exploitation of two weaknesses:

- Weak file processing step
- Certain math operations in the first 20 rounds have unexpected security issues

In the world of research and academic cryptography, an attack that has less computational complexity than a brute-force search is considered a break. However, academic research is sometimes far removed from practical application. The vulnerabilities identified in SHA-1 do not necessarily mean that the attack can be practically exploited. That being said, if an attacker could harness a massive distributed Internet search, theorists believe that finding a collision for SHA-1 would be possible.

In a practical sense, the primary concern about attacks that have been launched against SHA-1 is that they might pave the way to more efficient attacks—ones that might be practically carried out by would-be attackers. It is because of this potential that cryptographers believe that a migration to stronger hashes would be prudent. That is one reason why NIST is working toward a new solution.

A collision attack, as described earlier, does not present the same kinds of risks that a preimage attack does. A preimage attack on a cryptographic hash represents an attempt to find a message with a specific hash value. This type of attack differs from a collision attack in that a fixed hash or message is the focus of the attack.

Current application of cryptographic hashes for such purposes as password storage and document signing is only minimally affected by a collision attack. For instance, where it is used to sign a document, an attacker could not simply fake a signature from an existing document. In a case such as this, the attacker would also have to fool the private key holder

into signing a preselected document. Where it has been applied for password usage, practical security threats are also less likely. An attacker would not be able to reverse password encryption to obtain a user's password to use elsewhere using a collision attack. Constructing a password that works for a given account requires a preimage attack, along with access to the hash of the original password (typically held in a *shadow* file).

Usage of SHA-1

SHA-1 and other SHA hash algorithms (SHA-224, SHA-256, SHA-384, and SHA-512) are secure hash algorithms required by law for use in certain U.S. government applications. This includes the use of SHA-1 within other cryptographic algorithms and protocols to protect sensitive yet unclassified information. Adoption and usage of SHA-1 by private and commercial organizations has been encouraged by FIPS PUB 180-1. Central to the publication of SHA was the Digital Signature Standard, in which it is incorporated. We will examine this in the section "Exploring the Digital Signature Standard."

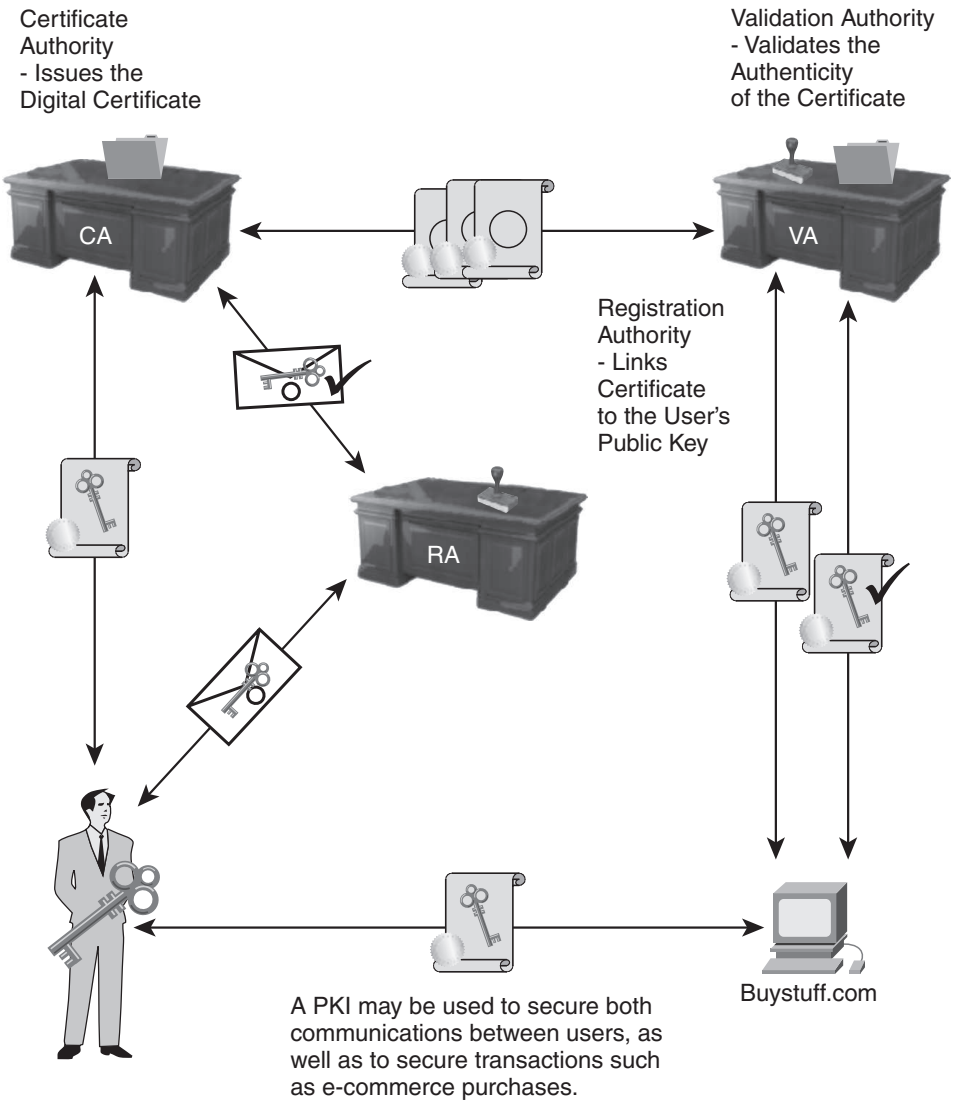
Using Digital Signatures

Much like written signatures, digital signatures may be used to authenticate an associated input. In the written sense, we might find a signature providing authentication on anything from a letter to a legal contract. In the digital sense, a digital signature's input is called a "message." These messages may be anything. They might be an e-mail or a legal contract, or it might even be a message sent in a more complicated cryptographic protocol.

These digital signatures are used to create public key infrastructure (PKI) schemes wherein a user's public key (whether for public key encryption, digital signatures, or another purpose) is linked to a user by a digital identity certificate issued by a certificate authority (CA). These PKI schemes seek to create an unbreakable bond between the user's information (such as name, address, and phone number) and the public key. This relationship allows the user to use these public keys as a form of electronic identification.

Let's look at an example. Figure 13-6 shows the process of using a digital signature to support a PKI scheme in which the user's public key is linked to the user through the digital certificate issued by the CA.

Figure 13-6 *PKI*



Here is how the process works:

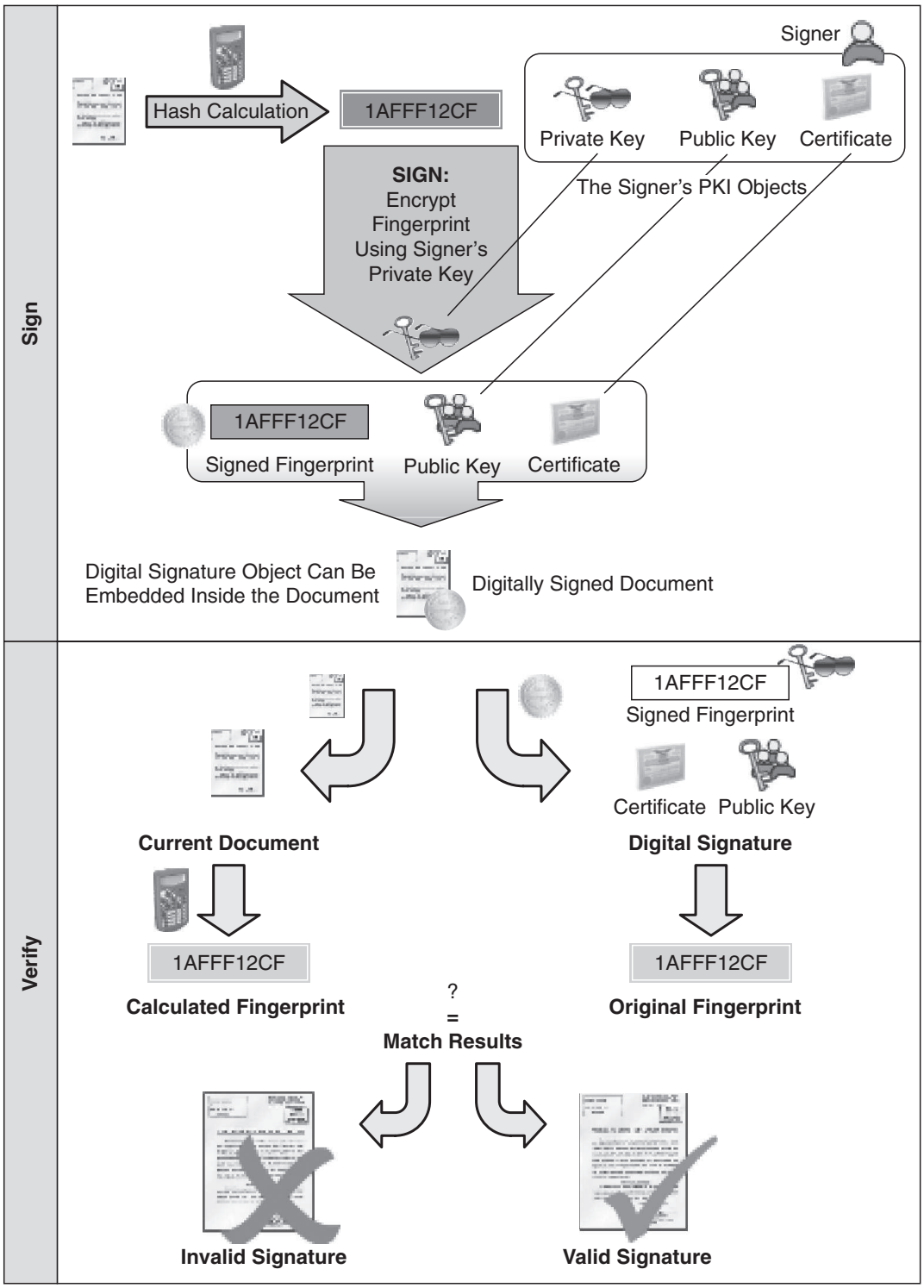
1. The user's identity is bound to his public key through the CA. This is carried out through the binding and issuance process.
2. The actual binding of the certificate is done by the registration authority (RA).
3. After the binding, the user may use this certificate to represent himself and in the creation of messages.
4. The user may freely distribute his public key to those with whom he wants to communicate.
5. To provide authenticity of a message, the user may sign his message using his private key. Recipients may then validate the message's authenticity by applying the sender's public key.
6. To secure communications with the holder of the public/private key pair, the user can encrypt a message with the recipient's public key. This message then may be decrypted only through the use of the recipient's private key.

A digital signature (or digital signature scheme) is a form of asymmetric cryptography that is used to simulate the security characteristics of a written signature in digital form. Digital signature schemes typically use two algorithms that employ a pair of public and private keys. One algorithm is used for signing, which involves the user's secret or private key. The other is used to verify these signatures. This typically involves the use of the user's public key. The end result of this signature process is called the digital signature. It has widespread use in electronic security.

Understanding Digital Signatures

To understand digital signatures, we need to begin by examining digital signature schemes and their commonalities. All digital signature schemes have a number of prior requirements. Figure 13-7 shows a digital signature.

Figure 13-7 *Digital Signature*



The first requirement is quality algorithms. As we have discussed, some of the available public key algorithms have been called into question with regard to security. Others are known to be insecure based on predictable attacks having been launched against them.

The second requirement is quality implementations. What this means is that even if you have a quality algorithm, if it is implemented incorrectly, it won't help you.

The third requirement is that the private key must remain secret. If this private key is compromised, an attacker can create an exact digital signature of anything he wants.

The fourth requirement is that the distribution of public keys has to be done in a manner that ensures that a public key belonging to a given user actually does belong to that user. Often this is done using a PKI. The public key user association is attested to by the operator of the PKI, the CA. In the case of "open" PKIs—ones in which anyone can request such an attestation—embodied in an identity certificate, the potential for mistaken attestation is not trivial. Unfortunately, commercial PKI providers have suffered a number of publicly known issues. Mistakes such as these could lead to falsely signed, and thus improperly attributed, documents. Maintaining a "closed" PKI system is more costly for organizations but less easily subverted, providing a stronger level of security for those who can take such steps.

Finally, beyond the PKI infrastructure and the steps that administrators must take to provide security, the fifth and final area of concern is users. The users of these PKI systems themselves (and their software) must take care to carry out the signature protocol properly to not compromise the signature.

All the conditions just listed must be met for a digital signature to reliably provide evidence of who sent the message, and therefore of his assent to its contents. Even legal measures cannot alter this reality.

Based on local laws, many countries accord a digital signature the same status as a traditional pen-and-paper signature with regard to its capacity to bind parties in a legal agreement such as a contract. Because of the legally binding nature of digital signatures in some parts of the world, it is generally best to use separate key pairs for encrypting and signing. Use of these key pairs allows an individual to engage in an encrypted conversation about matters that might be legally binding, such as the negotiation of a contract for employment. When the parties involved in the discussion reach an agreement, they can use their signing keys to "sign" the electronic document. At this point they are legally bound by the terms of a specific document. After this signing has taken place, the electronic document can be sent across an encrypted link to complete the transaction.

Digital Signature Scheme

Three algorithms generally make up a digital signature scheme:

- The key generation algorithm, which is used to randomly produce the key pair (public/private keys) used by the signer
- The signing algorithm, which, upon input of a message and a signing key, produces a signature
- The signature verifying algorithm, which, upon input of a message, a verifying key, and a signature, is used to either accept or reject the signature

Authentication and Integrity

One of the more practical uses of a digital signature in today's networks is for authentication and integrity checking. An example of this is the verification of authenticity in a message sent across a network.

Many times messages sent across the network include information about the entity sending the message. However, the authenticity of that information might be called into question. Digital signatures give us a mechanism to authenticate the source of such messages. With digital signatures, it is assumed that the ownership of a digital signature secret key is known to only a specific user. That being the case, a valid signature reflects that the message was sent by that specific user. The need for high levels of confidence in these matters is underscored by their use in financial matters, such as conducting credit card transactions or accessing bank account information.

For example, if a user accesses her bank account online and requests a transfer of funds between accounts, the bank must be certain of the authenticity of this request. If the bank has any question about this, transferring the funds would be a mistake.

In scenarios such as bank transactions, in addition to being assured of the user's authenticity, both the user (the sender) and her bank (the recipient of the message) may require confidence that the message has not been altered during transmission. Although the use of proper encryption can hide the contents of a message in transit, it may be possible to alter an encrypted message without understanding it. Certain nonmalleable encryption algorithms prevent this, but others do not. However, if a message is digitally signed, any change to the message while it is in transit invalidates the signature and alerts the parties that the message has been corrupted.

Examining RSA Signatures

The first algorithm found to be suitable for signing as well as encryption was RSA. RSA is an algorithm for public key cryptography. It represents what many believe to be one of the

first great advances in public key cryptography. Today RSA is widely used in a number of electronic commerce protocols. Thanks to its long cryptographic keys and the use of up-to-date implementations, it is believed to be secure.

Exploring the History of RSA

The algorithm that would become RSA was first described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman of MIT. A quick look at the first letters of their surnames tells you where the term RSA came from. Today RSA encryption is widely known and widely used for a variety of security needs.

Interestingly, a British mathematician named Clifford Cocks, who worked for the UK intelligence agency GCHQ, described an equivalent system in a top-secret internal document in 1973. However, because of the cost of computing resources to implement it, it was never deployed. This discovery was not revealed until 1997 because of its top-secret classification. Rivest, Shamir, and Adleman devised RSA independently of this initial work. In 1983 MIT was granted U.S. patent 4405829 for a "cryptographic communications system and method" that used the RSA algorithm. The patent expired on September 21, 2000.

Understanding How RSA Works

RSA uses a public key/private key combination. The public key in this pair can be known by anyone and can be distributed widely without issue to encrypt messages. After a message has been encrypted with a specific public key, it may be decrypted only through the use of the matching private key, which is privately held and is never distributed publicly.

If you are ready for some math, let's look at how the keys for the RSA algorithm are generated.

We begin by selecting two distinct large random prime numbers p and q. Next we compute $n=pq$, where n is used as a modulus for both the public and private keys. Then we compute the totient using the formula $\phi(n)=(p-1)(q-1)$.

Next we choose an integer e such that $1<e<\phi(n)$, and e and $\phi(n)$ share no factors other than 1 (coprime). The public key exponent that is released is e. You compute d to satisfy the congruence relation as $de=1 \pmod{\phi(n)}$. The private key exponent is d.

The public key consists of the modulus n and the public (or encryption) exponent e. The private key consists of the modules n and the private (or decryption) exponent d that must always remain secret.

Encrypting and Decrypting Messages with RSA

Based on our earlier discussion, you know that RSA employs a combination of a public and private key to both encrypt and decrypt messages. This means that a user who wants to send and encrypt a message with RSA would transmit her public key (n and e) to another user while keeping her private key to herself. When the recipient of the public key wants to communicate with the sender, he uses the public key he received from the sender. The way this works is that the user (through an application) first turns her message (m) into a number where $m<n$ by using an agreed-upon reversible protocol called a padding scheme. Next, the ciphertext c is computed such that $c=m^e \bmod n$. Finally, the user transmits this ciphertext to the holder of the private key—the only one who can decrypt this message.

Now that you have a sense of how encryption works, let's take a look at the decryption process. From the preceding example, the message recipient (holder of the private key) now must recover the message (m) from the ciphertext by using his private key exponent . He does this by using the computation $m=c^d \bmod n$. Given that the user has the message (m), she may now recover the original message.

Signing Messages with RSA

In addition to encrypting and decrypting messages, RSA can be used to sign messages. To continue our example of two individuals exchanging a message, let's suppose that one of the individuals uses the other's public key to encrypt and send a message. Although the message is encrypted, the sender may not be who we think she is.

In other words, because the public key has been widely distributed, you have no way of verifying that the message is in fact from the individual who claims to have sent it. That's because anyone could use this key to encrypt and send back a message to the recipient, who holds the private key. To be sure of the sender's identity, RSA may also be used to digitally sign the message as well as encrypt it. This ensures both the privacy of the message contents and the validity of its origin.

Let's take a look at the digital signing process with RSA. We can begin with a user who wants to send a signed message to a recipient:

1. The user who wants to send a signed message first must produce a hash value for the message itself and then raise it to the power of $d \bmod n$. (This same process is followed when a message is decrypted.)
2. The hash value is attached to the message as a digital signature.
3. When the message is received, the recipient must raise the signature to a power of $e \bmod n$, just like when the message contents are encrypted.
4. The resulting hash value is compared with the message's actual hash value to make sure they match.

5. If the two hash values are identical, the recipient can be assured that the author of the message does in fact have a secret key, so the message has not been tampered with.

Vulnerabilities of RSA

RSA's strength and security are based on two mathematical problems: the problem of factoring large numbers and the RSA problem. Cryptographers believe that complete decryption of an RSA ciphertext is not feasible based on the assumption that both of these problems are extremely difficult. In other words, currently there exists no efficient algorithm for solving them. Partial decryption is another matter. To provide security against this, it may be necessary to add a secure padding scheme. Even with these factors, a number of specific attacks still focus on RSA. The most common of these are listed in Table 13-2.

Table 13-2 *RSA Attack Vulnerabilities*

Attack	Description
Timing attack	In 1995 an attack against RSA was described wherein if the attacker knew a user's hardware in enough detail, and he could measure the decryption times for several known ciphertexts, he could deduce the decryption key quickly. This same attack could then also be applied against the RSA signature scheme as well. One way to defend against this form of attack is to make sure that a consistent amount of time is required for the decryption operation of each ciphertext. Although this would work, it may not be worth the performance degradation that would result. Most RSA implementations use an alternative approach known as blinding. In this approach, the multiplicative property of RSA is used. The result of applying RSA blinding is that the decryption time is no longer correlated to the value of the input ciphertext, so the timing attack fails.
Adaptive chosen ciphertext attack	The first practical adaptive chosen ciphertext attack against an RSA-encrypted message was described in 1995. This attack used the targeted flaws in the PKCS #1 scheme, which was used in concert with RSA. This attack focused on RSA implementations of the Secure Socket Layer protocol and was used to recover session keys. Because of the success of this attack, it is now recommended that RSA be used with other, more secure padding schemes, such as Optimal Asymmetric Encryption Padding. Additionally, RSA Laboratories has released updated versions of PKCS #1 that are not vulnerable to this form of attack.
Branch prediction analysis (BPA) attack	A number of processors use a branch predictor to determine whether a conditional branch in a program's instruction flow is likely to be taken. Generally speaking, these types of processors also implement simultaneous multithreading (SMT). A branch prediction analysis attack uses a spy process to statistically discover the private key when it is processed by these processors.

Exploring the Digital Signature Standard

NIST created DSS, specified in FIPS 186 [1], and adopted it in 1993. Since its adoption, the standard has been revised in 1996 and in 2000, with a degree of expansion. DSS employs the Digital Signature Algorithm (DSA), which was proposed for use in the standard by NIST.

Using the DSA Algorithm

The DSS outlines the use of the DSA by a signer to generate a digital signature to be applied to data and by a recipient of the data to verify the authenticity of the signature. To create the digital signature, you need both a public key and a private key. The private key is used to generate the signature, and the public key is used to verify the signature. For both signature generation and verification, the data, which is called a message, is reduced through the use of SHA.

If an individual does not know the private key of the message signer, he cannot generate the signer's correct signature. This prevents the digital forgery of these signatures. The signature then can be verified by the message recipient or anyone holding the signer's public key. Of course, for this system to be useful, we need a mechanism for associating public and private key pairs to the corresponding users. In other words, we must bind a user's identity and public key. The binding of this association may be certified by a mutually trusted third party.

We see the application of this in secure e-commerce. In these instances, a certifying authority signs the credentials containing a user's (or company's) public key and identity to form a certificate. The DSS does not go into detail about systems for certifying credentials and distributing certificates, because they are beyond the scope of the standard.