Chapter 13
Cryptography

THE FOLLOWING CEH EXAM TOPICS ARE COVERED IN THIS CHAPTER:

- ✓ **Cryptography**
- ✓ **Public key infrastructure**
- ✓ **Cryptography techniques**

If you are running across a web server that isn't encrypting all traffic, you have found an oddity. It's unusual for websites to be passing any traffic in the clear, even traffic that is brochureware. Web traffic is not the only type of network communications that is being encrypted. It's common for email servers to be using encryption as well. In fact, Transport Layer Security (TLS) is so well-known with so many implementations that it's easy to add it as a means to encrypt communications between systems using many protocols.

This is not to say that cryptography, the practice of hidden writing, is simple. There are many means of encrypting messages, and the idea of encrypting messages, or hiding the writing in messages, has been around for millennia. Probably almost as long as humans have been writing, there have been means to make it difficult for others to read what is written. Early forms of encrypted writing go back at least to the time of Julius Caesar, who developed his own means of converting a message from its original form into a form that enemies couldn't read.

Encryption principles give way to implementations of cryptography, like those that use asymmetric keys. This means there are two keys—one for encryption and one for decryption. There is also symmetric key encryption, which means there is a single key for both encryption and decryption. One of the essential elements of any cryptosystem is key management. This may

be handled through the use of a certificate authority, which issues encryption keys after, ideally, performing verification of identity. A certificate authority is not the only approach to handling the certificates, which hold the keys. There is also a decentralized approach, but in the end this is really about key management and identity verification.

Cryptography isn't only about confidentiality. Another property of most cryptosystems is integrity. You want to know not only that the data being sent was protected, but also that it came from the person you expected it to come from. Maybe most important, you want to make sure the data that was sent is the data that was received. Fortunately, we can use cryptographic algorithms to perform this sort of verification. A hash algorithm generates a fixed-length output from variable-length input. There are multiple hash algorithms to know about and understand.

Encryption is about privacy and confidentiality, but it's important to realize that encryption is not the solution to every problem. Whole-disk encryption is common. What whole-disk encryption protects is a dead hard drive (meaning it isn't in a powered-up system). No one can steal an encrypted hard drive and be able to read the contents. If an attacker can gain access to a system through either malware or stolen credentials and the malware is running in the context of a legitimate user with permissions or the stolen credentials have permissions on the system, the disk may as well not be encrypted. A running system with a logged-in user appears to be unencrypted to that user. The same is true with mobile devices with encrypted storage. This is why it's important to recognize what encryption can do and can't do.

Basic Encryption

We start at the beginning, which is the word. Or, more accurately, the beginning is plain text. Plain text is what you are reading. It is text, or any other type of data, that is consumable without anything else being done to it. It hasn't been transformed in any way. Once this plain text has gone through the transformation process from a cryptographic algorithm, it is called *ciphertext*. Ciphertext should be unreadable on its face and require a process to reverse it back to plain text.

Substitution Ciphers

Not all encryption has to use the mathematical processes we use today. The goal is simply to generate something that someone who intercepted the message wouldn't be able to read without some additional information. An early method of cryptography is the rotation cipher. This takes an alphabet and rotates the letters by some value. When you place the original alphabet over the top of the rotated alphabet, you can easily convert from one to the other. If your original alphabet is on top, encryption is just taking the letter directly below. This is called a *substitution cipher*, because one letter is simply substituted for another in a fixed, known way. Here is an example of what this would look like:

Rotation Cipher

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EFGHIJKLMNOPQRSTUVWXYZABCD
```

To encrypt a word like *hello*, you would find each letter in your word in the top row and find its substitute in the bottom row. This would give you *lipps* as the resulting ciphertext. To decrypt, you find each letter in the bottom row and replace it with the corresponding letter in the same position in the top row. It is said that this technique of encrypting messages was used by Julius Caesar, who used it to send messages to his generals in the field. If the messages were intercepted, they wouldn't be readable without the key. In this case, the key is just the number of positions the alphabet is to be rotated.

Users of the former newsgroup network Usenet commonly used a rotation cipher to obscure anything in a message that might be considered offensive or a spoiler. This meant that to read it, someone would have to deliberately decrypt that portion of the message and couldn't complain that it was in the clear. The common rotation was referred to as *rot13*, meaning the text was rotated halfway around the alphabet, 13 positions.

This is not to say the key couldn't be derived. This sort of cipher can have the key revealed through a brute-force attack, meaning every rotation possibility is tried (in the Latin alphabet that we use, there are 25 possibilities, since 26 would just rotate back to where you started and not be any rotation at all). Eventually, a message resulting from these attempts would make sense, at which point you would know the key.

Another possibility for deriving the key in a substitution cipher is to use frequency analysis. A frequency analysis shows the statistical distribution of letters used in a language. Figure 13.1 shows the English language distribution. The most used letter, based on this distribution, is *e*, followed by *t* and then *a*. A frequency analysis on a substitution cipher like the rotation cipher finds the most used letters in the ciphertext and applies statistical probability based on a normal distribution. Once you start substituting the most commonly used letters, some of the words will start to make sense, so in cases where letters have the same distribution probability, you may be able to identify letters based on the words that are clear in the partially decoded ciphertext. Of course, with a rotation cipher, once you have one letter decoded, you will have the key because you know how far away the ciphertext letter is from the plaintext letter.

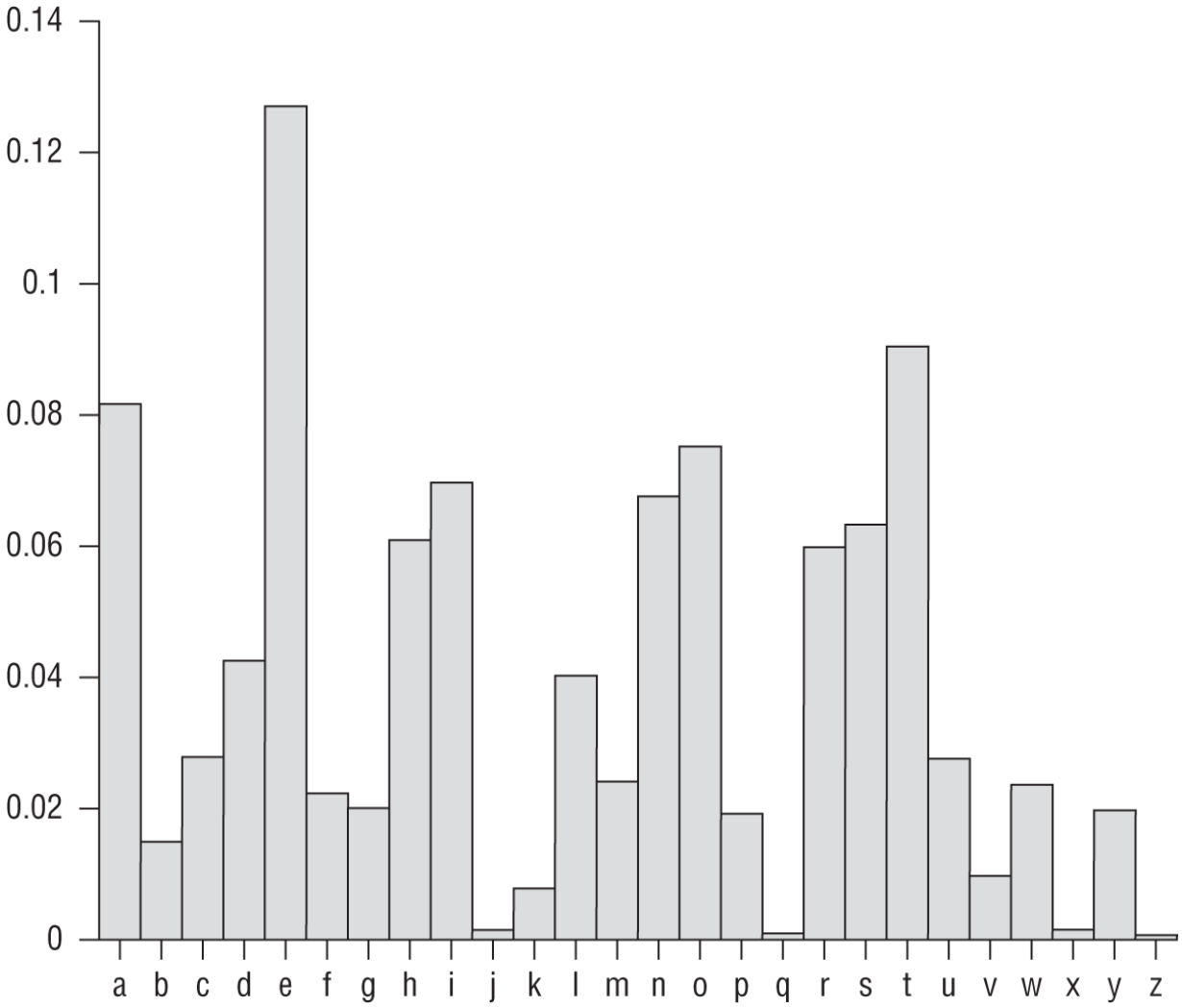

FIGURE 13.1 English letter normal distribution

Because the rotation cipher is easily decrypted, especially if it's known to be a rotation cipher, there are other methods of dealing with encryption by hand. One method uses a multidimensional approach. Instead of just laying one alphabet over the top of a rotated alphabet, you can use a grid. Once you have the grid, you need a more complex key. In this case, you can use a word. This word becomes the method that is used to convert the plain text into the ciphertext. This type of cipher is named after its developer, Blaise de Vigenère. When you use this sort of polyalphabetic cipher, you are using a Vigenère cipher.

You can see an example of the grid or square that you would use to encrypt messages using the Vigenère cipher in Figure 13.2. This process works by taking plain text and a word used as a key. Let's say you want to use the key

hello and encrypt the word *deforestation*. You'll notice there aren't the same number of letters in the key as there are in the plain text. This means you repeat the key over and over until you run out of letters in the plain text. In this case, the key becomes *hellohellohel* to match the 13 letters in *deforestation*. The key letter is matched on the top row with the letter from the plain text matched along the left column. Where the row and column intersect, you have the letter to be used in the ciphertext. For this example, the ciphertext you would end up with would be *kiqzflwelhpsy*.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

FIGURE 13.2 Vigenère square

Another way to think about this is to align the key along the x-axis while the plain text is along the y-axis. You would start with the *d* along the y-axis and the *h* along the x-axis. The point at which these two intersect is *k* in the grid. For the second letter, you look up *e* along the y-axis and *e* along the x-axis and find where the row and column intersect and find *i*. You keep going with this process until you have run out of letters in the plain text.

You will note how time-consuming this process can be. Fortunately, today, there are easy ways to do this encryption and decryption. There are programs available and websites that will encrypt and decrypt this sort of cipher for you, just as there are for the rotation cipher. This does mean, though, that while clever and challenging to decipher manually when they were developed, both these substitution ciphers are unusable today because modern computing has made them trivial to crack. As a result, we need to look at other methods of encryption, including and perhaps especially transformation ciphers.

Diffie–Hellman

As you might expect, key management is essential when it comes to cryptography. The keys are known in advance. They are sometimes called *pre-shared keys* because they have been shared in advance of their need. If the keys are not shared ahead of time, there needs to be a way for two parties in an encrypted communication to share keys without anyone being able to intercept them. Once a key is known, it's like the messages weren't encrypted to begin with. Ideally, the keys aren't exchanged at all. When keys are exchanged, there is the possibility they will become known. A better approach would be for both sides to derive the keys. This means the key is never exchanged.

Whitfield Diffie and Martin Hellman came up with an approach that would allow two endpoints to generate a key without ever transmitting any data that would allow the key to be known. While the details of it involve a lot of math that we won't go into here, it's easy enough in the abstract to understand. Figure 13.3 shows how keys are generated with Diffie-Hellman, using paints to demonstrate how the process works.

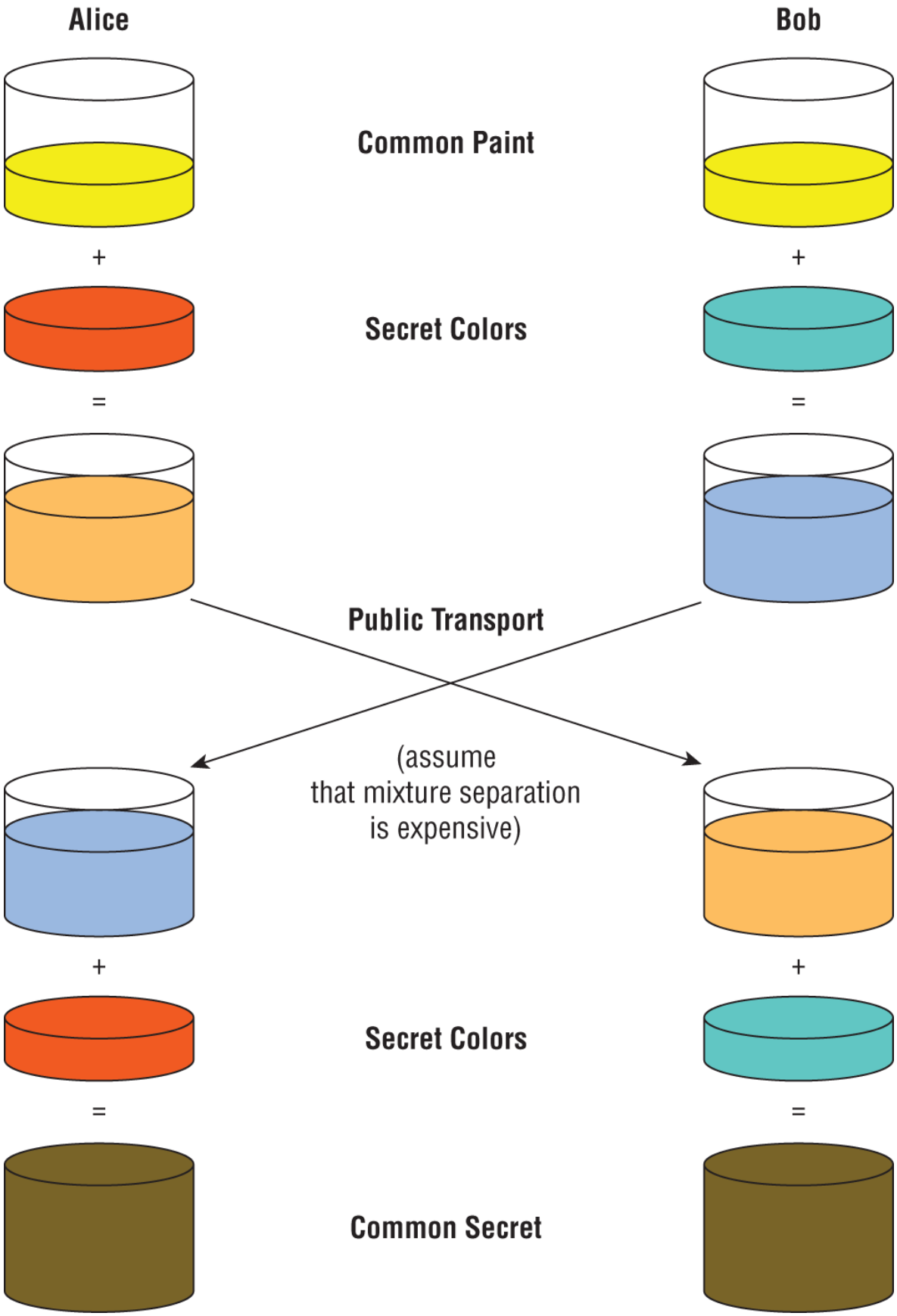
Alice
Bob
Common Paint
+
Secret Colors
=
Public Transport
(assume
that mixture separation
is expensive)
+
Secret Colors
=
Common Secret

FIGURE 13.3 Diffie-Hellman process

Both sides of the conversation start with the same base paint (initial key value). In Figure 13.3, this base paint "value" is green. Each side adds to that base a random value (or paint color in this case). Alice adds red, and Bob adds yellow. This results in orange for Alice and blue for Bob. The two sides have entirely different values that result from the initial injection of random values. At this point, the two sides can exchange their results. There is nothing in this exchange that will give up anything to an attacker. This assumes that separating the colors out to the two colors (values) that created the transmitted color is a very expensive process. This would be a mathematical algorithm that can't be reversed to the two factors.

Once both sides have the value/color from the other side, all they need to do is add in the random value they added to the base color. This means both sides now have the base color + their random value + the other side's random value. They both have the same color/value, which can be used as the key. Figure 13.3 shows that adding the random colors in after the transmission results in a common secret/color that is brown. Using this process, you can create a key that can be used to encrypt and decrypt messages from both parties.

Symmetric Key Cryptography

The key that comes out of the Diffie-Hellman exchange is a symmetric key. We know this because the same key is used to encrypt and decrypt messages. Since it's the same in both directions, it's symmetric. This is a fairly common approach to keys because there is a single key that needs to be managed and it doesn't matter which direction the transmission is going. Both sides are expected to have the same key. If they don't, the message that was encrypted by one key won't be decrypted by another key and will look like gibberish.

Any symmetric key algorithm can be either a stream or a block cipher. Block ciphers take the entire block of data to be encrypted and turn it into fixed-length blocks. If the total length of the data isn't a multiple of the block size, the last block is padded to get to the size of the block. A stream cipher, on the other hand, encrypts the data byte for byte. Think of the

Vigenère cipher as an example of a stream cipher. The data is encrypted one letter at a time without any reliance on any other portion of the message. A block cipher may commonly use a block length of 64 bits, which would be 8 single-byte characters.

Data Encryption Standard

The Data Encryption Standard (DES) is a block cipher that uses a symmetric key. This is a long-deprecated encryption standard, but it raises an important element about cryptography. One of the problems with DES is that it only uses a 56-bit key. When it was approved in the 1970s, based on a cipher named Lucifer from IBM, computing power was not strong enough to take on a long key length. A 56-bit key was all that could be accepted. As computing power has increased, even without issues in the algorithm itself, the key becomes vulnerable to brute-force attacks. This means someone trying to crack an encrypted message can try every possible key against the ciphertext in order to extract plain text. The block size used for DES was 64 bits, though the key is only 56 bits. That's because 8 bits of the key are used for parity.

When it became clear that DES was vulnerable, for a couple of reasons, the National Institute of Standards and Technology (NIST) launched a search for a replacement algorithm. Unfortunately, the search wasn't going to be fast enough to protect data being encrypted with DES. As a result, a temporary measure was created to increase the strength of DES. The same algorithm could still be used, but it could be applied multiple times. The problem is if you keep encrypting with the same key over and over, you can still attack the key if there are weaknesses there. Instead, the short-term workaround was to use multiple keys. Even that's not sufficient, however.

With Triple DES (3DES), there are three keys, which increases the effective key length to 168 bits. However, it's not a single 168-bit key. It's three keys applied one at a time. One key is used to encrypt the message. The second key is used to decrypt the message. No need to worry here. Since you are decrypting with an entirely separate key, you still end up with ciphertext and not plain text. You take the final key and employ an encryption algorithm again. This results in a third, entirely different ciphertext. Essentially, you take plain text to get ciphertext once, then take ciphertext,

decrypt it into ciphertext, and take that resulting ciphertext and encrypt it to obtain one more round of ciphertext.

The reason for mentioning DES and 3DES, in spite of the fact that both are deprecated, is to suggest that key strength is important. However, you can't necessarily compare one key length with another. Since three 56-bit DES keys total 168 bits, you might think you're working with a 168-bit key and any other symmetric algorithm using a shorter key would be worse. There are two things to think about here. The first is that the algorithm is the important part. It doesn't matter how large the key is if the algorithm is weak. You could double the key length or even triple it, but the algorithm could still be broken to allow the message to be decrypted. The second is that the 3DES key isn't really 168 bits. Instead, it's three separate keys.

Advanced Encryption Standard

The replacement algorithm for DES was the Advanced Encryption Standard (AES). Just as Lucifer was used to create the standard known as DES, the Rijndael cipher was used as the basis for the AES. The cipher is different from the standard, and while Rijndael was developed earlier, the standard was published in 2001. AES is a block cipher that uses multiple key lengths and a block length of 128 bits. When AES was selected, three different key lengths were also selected. The first key length used was 128 bits, but key lengths of 192 bits and 256 bits are also possible. Just as DES with a 56-bit key became vulnerable over time, AES has been shown to have theoretical vulnerabilities to attack with a 128-bit key length and the industry has generally moved to longer key lengths. Computing power is not the barrier that it once was, so longer key lengths are much easier to support.

One thing to be aware of is that an encryption cipher is only a portion of what is necessary to allow for messages to be encrypted between endpoints. There are multiple components, and all of them together are called a *ciphersuite*. The first element is the one used to exchange the key. This may commonly be DH, though it isn't always. The second is the encryption cipher to be used. Finally, there is a message authentication code. We'll cover that in more detail later. In the following code listing, though, you can see a list of the ciphersuites that are allowed on Google's web server. This list was created with `sslscan`, a tool used to identify supported

cryptographic ciphers on servers that use the Secure Sockets Layer (SSL) and TLS protocols.

Ciphersuites

```
   Supported Server Cipher(s):
 Preferred TLSv1.3  128 bits  TLS_AES_128_GCM_SHA256
Curve 25519 DHE 253
 Accepted  TLSv1.3  256 bits  TLS_AES_256_GCM_SHA384
Curve 25519 DHE 253
 Accepted  TLSv1.3  256 bits  TLS_CHACHA20_POLY1305_SHA256
Curve 25519 DHE 253
 Preferred TLSv1.2  256 bits  ECDHE-ECDSA-CHACHA20-POLY1305
Curve 25519 DHE 253
 Accepted  TLSv1.2  128 bits  ECDHE-ECDSA-AES128-GCM-SHA256
Curve 25519 DHE 253
 Accepted  TLSv1.2  256 bits  ECDHE-ECDSA-AES256-GCM-SHA384
Curve 25519 DHE 253
 Accepted  TLSv1.2  128 bits  ECDHE-ECDSA-AES128-SHA
Curve 25519 DHE 253
 Accepted  TLSv1.2  256 bits  ECDHE-ECDSA-AES256-SHA
Curve 25519 DHE 253
 Accepted  TLSv1.2  256 bits  ECDHE-RSA-CHACHA20-POLY1305
Curve 25519 DHE 253
 Accepted  TLSv1.2  128 bits  ECDHE-RSA-AES128-GCM-SHA256
Curve 25519 DHE 253
 Accepted  TLSv1.2  256 bits  ECDHE-RSA-AES256-GCM-SHA384
Curve 25519 DHE 253
 Accepted  TLSv1.2  128 bits  ECDHE-RSA-AES128-SHA
Curve 25519 DHE 253
 Accepted  TLSv1.2  256 bits  ECDHE-RSA-AES256-SHA
Curve 25519 DHE 253
 Accepted  TLSv1.2  128 bits  AES128-GCM-SHA256
 Accepted  TLSv1.2  256 bits  AES256-GCM-SHA384
 Accepted  TLSv1.2  128 bits  AES128-SHA
 Accepted  TLSv1.2  256 bits  AES256-SHA
 Accepted  TLSv1.2  112 bits  DES-CBC3-SHA
```

You will see in that list that both AES-128 and AES-256 are supported. One other line to make note of is that DES is also supported on Google's web server. Although it's been deprecated in favor of AES, there are still browsers that run on systems that may be less capable. This means DES may still be necessary. While it's accepted, it's not one of the preferred algorithms. You may notice that DES here uses Cipher Block Chaining

(CBC). This is a way of further transforming the message. With CBC, each block of ciphertext is XOR'ed with the previous block. Since the first block in has no previous block, there is an initialization vector (IV) that is used to XOR against. Keep in mind that ciphertext is always binary, since each byte may not align with a printable ASCII character.

There are a limited number of attacks that have been proposed as potentially being successful against AES. One of these is a side-channel attack. A side-channel attack relies on using something other than a weakness in the algorithm. Instead, the implementation becomes the target. Information can be leaked as a result of power consumption or processor utilization, for instance. There may also be electromagnetic leaks that could provide information to an attacker. This is not the sort of attack that someone would be able to accomplish without extensive understanding of cryptography and how systems work. This does not mean that AES is entirely invulnerable, but so far, there have been no weaknesses discovered in the algorithm.

Other types of cryptographic attacks have been proposed, including a related-key attack. A related-key attack is where the attacker can observe ciphertext that has been encrypted using several different keys. While the attacker doesn't know what the keys are (otherwise, they could just use the key to decrypt the text), there is some mathematical relationship between the keys that is known to the attacker. It could be, for example, that a large number of bits for the keys are the same. This was proposed as an attack against AES in 2009 as a result of the key schedule used by AES. However, protecting against this attack comes down to the implementation. The implementation of AES can ensure related keys are not allowed to be handed out.

While there have been key recovery attacks against AES, where the key can be obtained to be used to decrypt ciphertext, they have shortened the amount of time it would take to brute-force the key, but there is still a significant amount of time and computing power required to get the key. Longer key lengths will help protect against this sort of attack. Essentially, there is currently no practical attack against AES.

Asymmetric Key Cryptography

Where symmetric key cryptography uses a single key for both encryption and decryption, asymmetric key cryptography uses two keys. Because of this, asymmetric key cryptography is sometimes called *public key cryptography*. One key is the public key, and the other is the private key. These two keys are tied mathematically so that what is encrypted by one key can be decrypted by the other. This is how we get around the issue of transmitting keys from one party to the other. If there are two keys, only one needs to be protected. The other doesn't need any protection and can be transmitted safely. In fact, the public key is meant to be public, meaning the only way public key cryptography works is if people actually have the public key.

Public key encryption uses the public key to encrypt messages that only the private key can decrypt. The private key is the only key that needs to be protected in this scheme, which is fine because it is only needed to decrypt messages that have been sent using the corresponding public key. If you want to exchange encrypted messages with someone, you should provide them with your public key, and they should provide you with theirs. There is no danger or risk associated with just sending these keys around. The only thing that can be done with them is to interface with the corresponding private key. There is no danger of anyone using the same public key to decrypt a message sent by someone else. The keys don't function in that way.

One common algorithm that uses public key cryptography is the Rivest-Shamir-Adleman (RSA) algorithm. This is an algorithm that uses a key based on a pair of large prime numbers. The key sizes used by RSA are 1,024 bits, 2,048 bits, and 4,096 bits. This is another area that highlights why key strengths can't simply be compared from one algorithm to another. Asymmetric algorithms behave completely differently. Just because AES uses 128 bits and RSA uses 1,024 doesn't mean RSA is an order of magnitude stronger. The two are very different, so the key sizes can't be compared directly. The only thing you can say about key sizes is that RSA with a 2,048-bit key is considerably stronger than RSA with a 1,024-bit key, just as AES with a 256-bit key is considerably stronger than AES with a 128-bit key.

Hybrid Cryptosystem

You may wonder why there are two types of algorithms—symmetric and asymmetric. Why not just use one? Even though you can't directly compare symmetric and asymmetric, asymmetric keys are considerably larger, and if an attack relies on brute force, it would take vastly longer to brute-force a 1,024-bit key as compared with a 128-bit key, simply because there are many orders of magnitude more keys in the key space. So, why not just use asymmetric key cryptography everywhere?

First, asymmetric key cryptography is expensive in terms of computing power. It's not especially fast because of the size of the key. While this isn't the problem that it once was because modern computers are so fast, we are still having to use larger and larger keys, which can mean more computation because of the key length. Second, there is a lot of overhead associated with public key cryptography. If we wanted to use public key cryptography everywhere, everyone would need to have such a key. This is largely impractical. You'll see why a little later when we get to key management and certificate authorities.

However, we still have the problem of key exchange when it comes to symmetric key encryption. We can fix that with public key encryption, though. We do that by using both public key and symmetric key encryption. This approach is referred to as a hybrid cryptosystem. The public key can be used to protect the symmetric key so there isn't a problem with anyone obtaining this symmetric key. The symmetric key in this case is sometimes called a *session key*, because it is used to encrypt the data in a session between two systems. This sort of hybrid cryptosystem is most commonly used when communicating with web servers.

While Diffie-Hellman may be used to negotiate the key derivation, in practice it isn't used in a hybrid cryptosystem. Instead, either side would generate a symmetric key. Once the key has been generated, the public key would be used to encrypt it. When it comes to web communications, generally the only side with a public key is the web server. This means the client would generate the session (symmetric) key and send it to the web server, encrypting it with the web server's public key.

Nonrepudiation

Another value of public key cryptography is that the public key and private key belong exclusively to a person or system, depending on its ultimate use. We can make use of the public key belonging to a known individual. In addition to encryption, asymmetric keys can be used to digitally sign messages. A message is signed using someone's private key. It can be verified by that person's public key. We don't have to worry about privacy in this case because it's just a signature. It's used to demonstrate that a message has been generated by the person who owns the private key. This principle of demonstrating that a message originated from the owner of a private key is called *nonrepudiation*.

In fairness, nonrepudiation says that when a message has been sent using a private key, the owner of that private key can't say the message didn't originate with them. This is sort of the reverse of how it was stated before. The two ways of thinking about it mean the same, or at least have the same effect. It's just that the term *nonrepudiation* means you can't go back on something by claiming it didn't come from you.

This does require that the private key be protected. If a private key is not protected in any way and it gets out into the open, it could be used by anyone. This is why protecting a key means different things. The first is that the file containing the key should have appropriate permissions so only the owner of the file can access it. This is only part of the protection, though. An attacker may still be able to gain access to someone's system as that user and obtain the key. This is why using the key requires a password. The password would be requested before the key could be used to sign the message or else the message signing would fail.

Elliptic Curve Cryptography

While the factoring of prime numbers has commonly been used for keying, because the power and time necessary to perform computations on these large numbers is extensive, using them is not the only way to generate keys. Complex problems like factoring of prime numbers are considered intractable, which means the power necessary to solve the problem is effectively prohibitive. Large prime numbers are simply time-consuming, however. Applying more computing power can solve these problems. This becomes easier as computing power becomes much cheaper. Computers have continued to become significantly more powerful, and it's far less

difficult to apply many of them at a time to take on complex and difficult tasks.

Elliptic curve cryptography (ECC), though, takes a different approach. Rather than just using resource-intensive computations such as prime number factorization, ECC uses discrete logarithms. It is assumed that identifying the discrete logarithm of a random elliptic curve element is not just computationally difficult but infeasible. When using ECC, the key size ends up being much smaller. This reduces the amount of computing power necessary for the encryption and decryption.

An elliptic curve can be seen by plotting the graph of a polynomial function. An example, as shown in Figure 13.4, is the plot of the function $y2 = x3 – x + 1$. A logarithm is the inverse of exponentiation, and a discrete logarithm is the value that you raise one number to in order to get another number. Essentially, it's the index. If you take the function $bk = a$, k is the discrete logarithm. So essentially, you select a point on the curve and find the value of k for that point. That is the value that is being used in ECC.

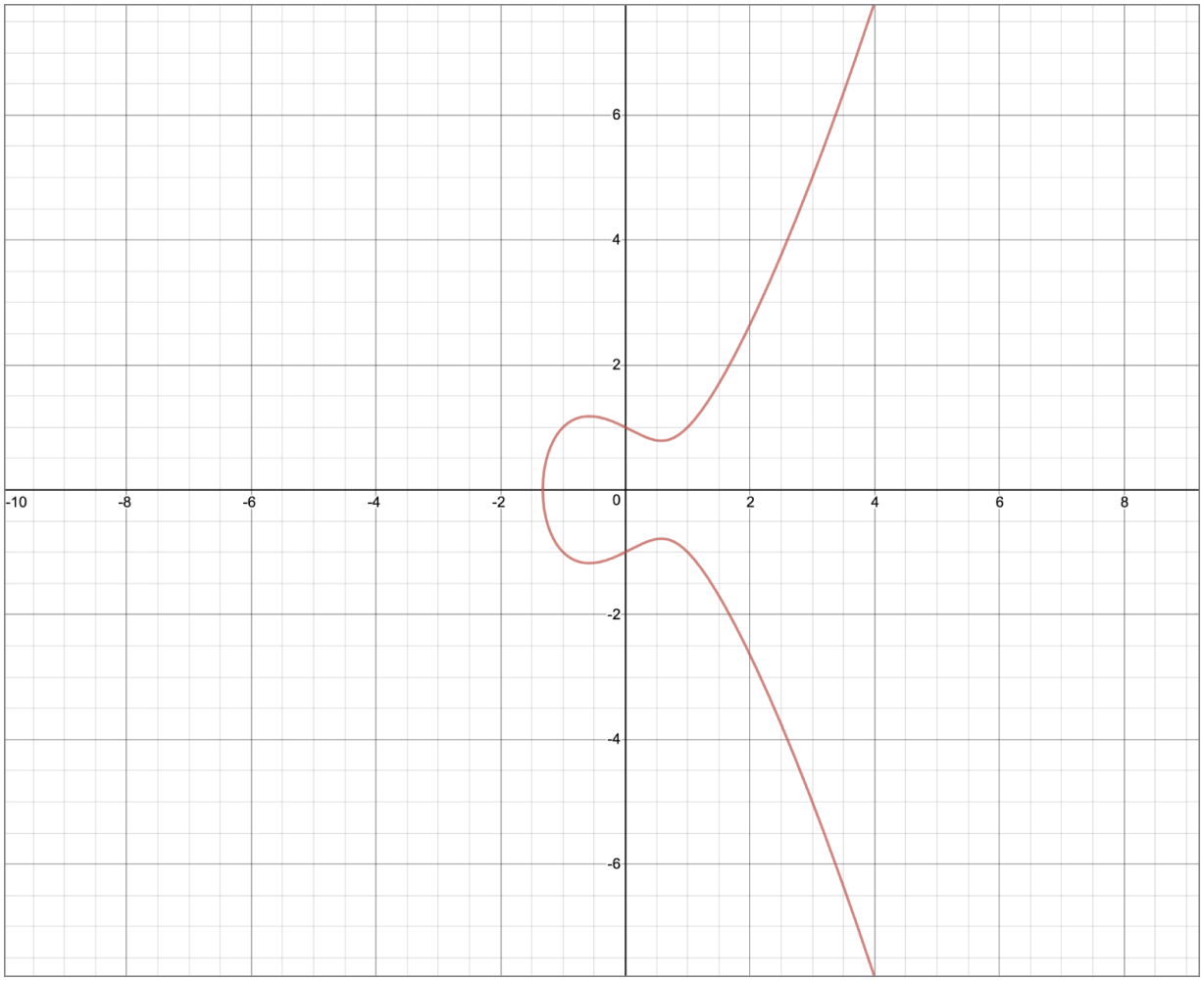

FIGURE 13.4 Elliptic curve

You may think this wouldn't be that hard. The reality is there is no standard approach to computing a discrete logarithm. There are some specific cases where they can be computed easily and quickly, but discrete logarithms are not a problem that has a generalized solution, so given discrete logarithm *x*, there is only one way to compute the solution. This is what makes ECC insoluble. Unlike factors of prime numbers, for which there are known computations, even if they are resource intensive, ECC works because there is no way to reverse the process in order to find the answer.

Perhaps an easier way to think about this is that there are a lot of multivariable polynomial equations. There is not a single algorithm that allows taking a point on the curve generated by the equation and reversing it to the original equation. Without that generalized algorithm that allows all equations to be solved (or solved in reverse to be more accurate), this is

another intractable problem that allows for the creation of keys that can't be determined easily.

You can see the use of ECC in the output earlier from the scan of the web server showing supported cryptographic schemes. When you see ECDH as part of the ciphersuite, that is Elliptic Curve Diffie-Hellman and is a key agreement protocol that uses a Diffie-Hellman variant that makes use of elliptic curve cryptography.

Certificate Authorities and Key Management

Keys make cryptography work. Without them, we don't have a way to reverse the encryption process. Transforming text into something unintelligible is easy. It's the transformation that can be reversed that is complex. Since having the key gives someone the ability to decrypt messages, protecting the key is essential. Keys that are used for sessions need to be protected but don't need to be stored beyond the sessions for which they are needed. Once the session is over, perhaps in a handful of minutes when a client is talking to a web server, the key is discarded. However, keys that have been created for a person and associated to that person, even if the person is a server, need to be persistent.

Keys can be stored inside a data structure called a *certificate*. The certificate structure is defined by X.509. X.509 is part of a larger X.500 standard used to define digital directory services. As part of a digital directory, encryption certificates can be stored. When you generate a public key, it gets stored in a certificate record. You don't want to have to manage your own certificate once you have it. Remember that the certificate includes the keying information, and when you are using public key cryptography, the point is to get your public key out there. This means ideally there is a repository to store certificates.

Certificate Authority

A certificate authority (CA) is a repository of certificates. It issues certificates to users, which means it collects information from the user and then generates the key to provide to the user. The certificate is stored in the authority and also provided to the user. Of course, someone has to manage

the certificate repository. The CA and related systems is called *public key infrastructure* (PKI). A company that has one of these repositories is sometimes called a *certificate authority* since it is the authority that manages the certificates. Sometimes the software itself is referred to as the *authority*. This is the case with a piece of software called SimpleAuthority. This software uses the OpenSSL library and utilities to generate the certificates and handles the storage, creation, and management through a graphical interface. Figure 13.5 shows creating a certificate authority, meaning creating the root certificate and the certificate repository, using Simple Authority. The root certificate is the certificate that all other certificates in the CA are signed by.

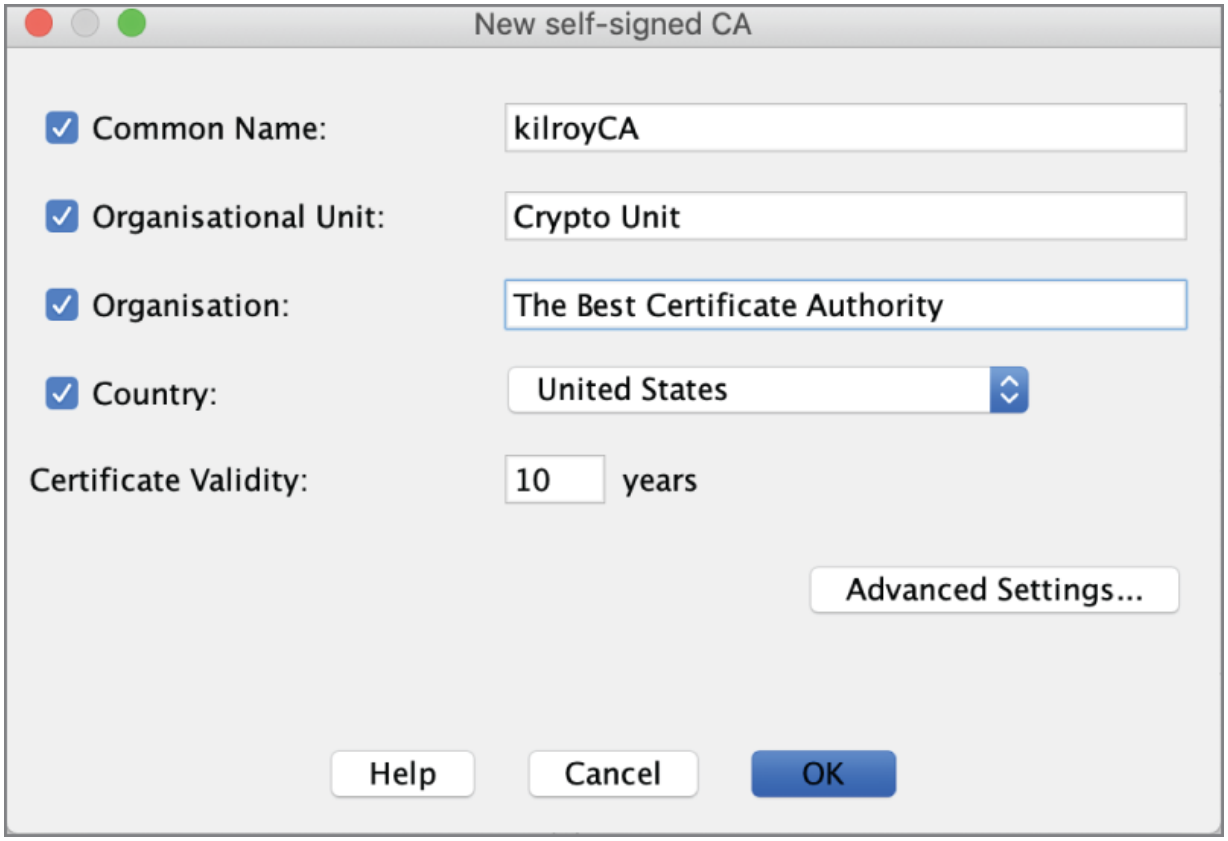

FIGURE 13.5 Simple Authority CA creation

Once the root certificate has been generated, you can create other certificates. Figure 13.6 shows not only the creation of a new certificate but also the different types of certificates that can be created. Any entity that wants to encrypt messages using public key cryptography needs to have a certificate. If you want to send messages to someone, you can get a

certificate. Servers, though, also need to have certificates. Any web server that uses encrypted messages, and these days it's rare to find a web server that isn't encrypting traffic, needs to have a certificate. You can see that as one of the options. If you were going to set up a web server that was going to use TLS, you would select SSL Server, since TLS is the current implementation of what was once SSL.

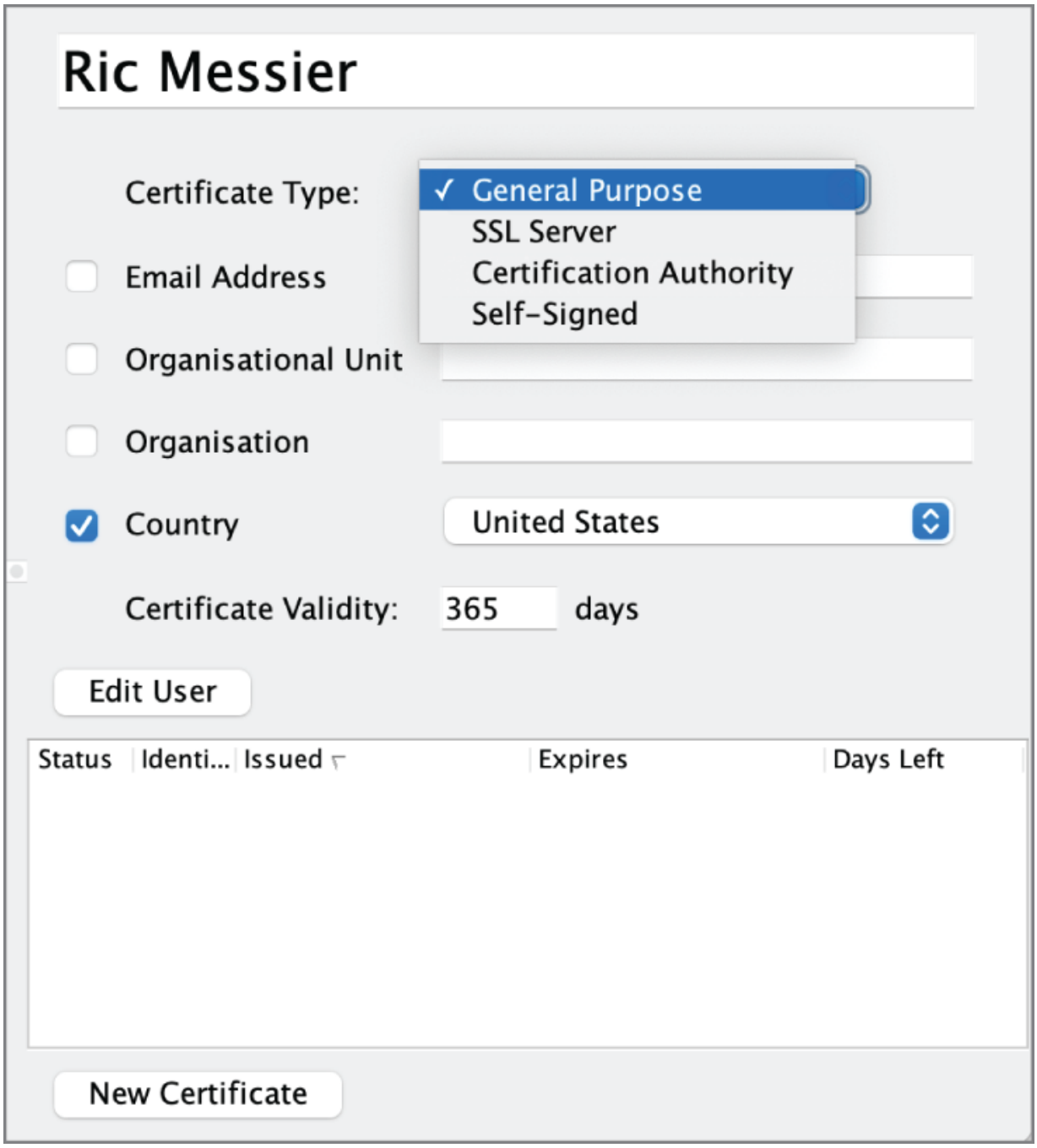

FIGURE 13.6 Certificate creation

Once you have provided the details necessary for the certificate, the CA will create the keys and store the certificate. The CA will then provide the means to obtain the certificate for the user. This is necessary since the user has to have the private key. The CA can provide the public key to anyone who wants it, but the user who owns the certificate needs to have the private key portion. Once the certificate has been created, you can open it to look at details. You can see these details in Figure 13.7. What you will see there are the details provided during the certificate creation as well as information about the key. What you can't see is the key itself. This is just a set of bits that would not be printable, since each byte is not limited to the set of values that are considered printable in the ASCII table. Any byte value is possible, and most of them would just look like gibberish.

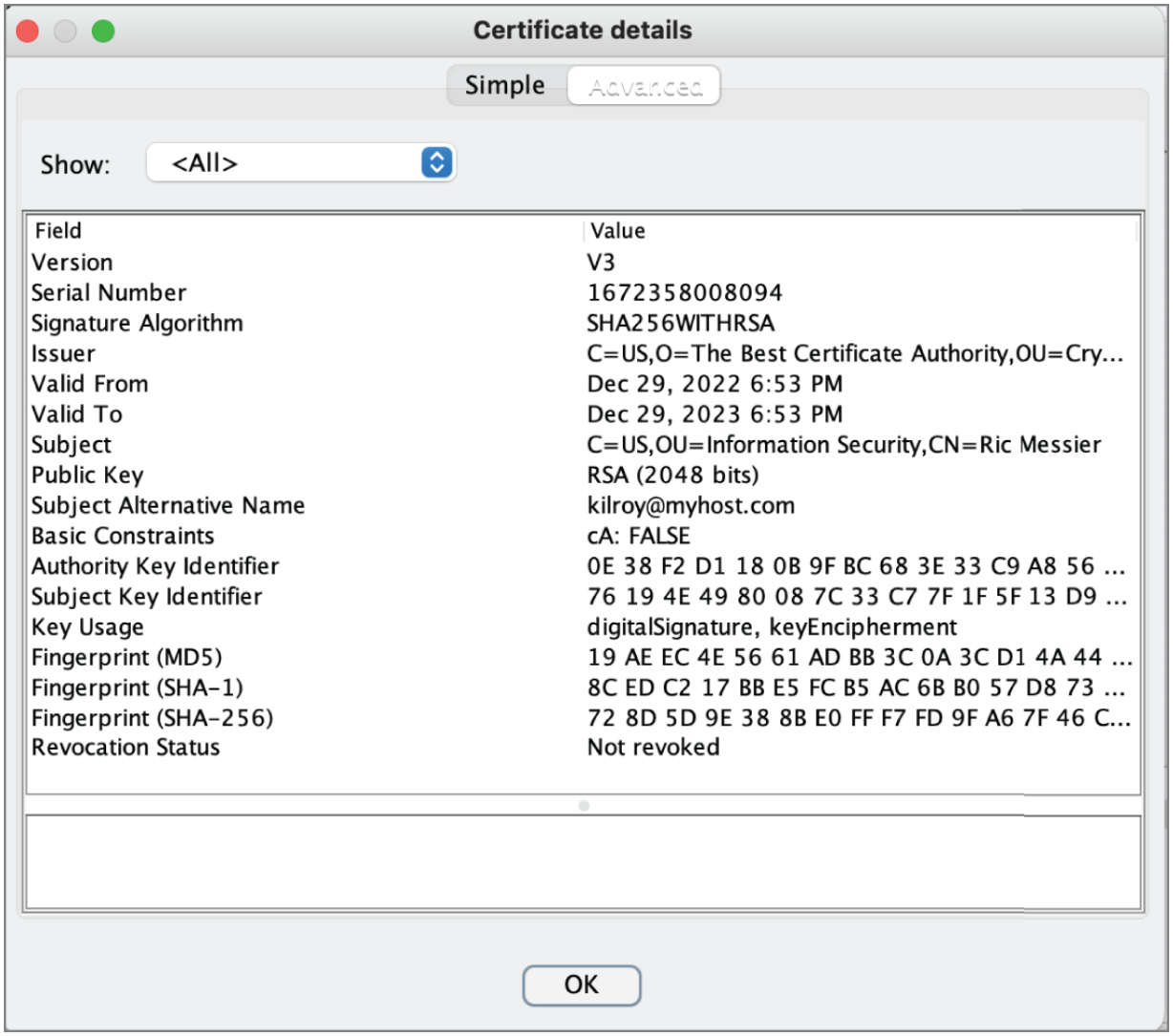

FIGURE 13.7 Certificate details

Instead of the key itself, which wouldn't mean much, you can see a fingerprint of the key. The fingerprint is a fixed-length value that is shown in hexadecimal. You could, of course, also show the key in hexadecimal rather than trying to generate characters from the byte values. A 4,096-bit key, though, would be 512 bytes. A byte is represented by two hexadecimal digits. That means showing a 4,096-bit key in hexadecimal would be 1,024 characters. Just to give you a sense of how much that is, this paragraph is 561 characters, including spaces and punctuation.

When you use a certificate authority, the authority can revoke certificates. This may be because a user is no longer associated with the organization that manages the authority, for instance. Enterprises may run their own authorities. Since there is identification information in the certificate, like an email address, if the email address is no longer valid, the certificate may need to be revoked. When a certificate has been revoked, any party validating the certificate against the authority should not accept the certificate. This is important since there is an element of identity associated with certificates. The certificate can provide a verified identity, which could be used to authenticate someone. If a certificate has been revoked, the user should not be trusted.

Revoked certificates are managed through the use of certificate revocation lists (CRLs). The problem with CRLs historically has been that they are not always requested to validate a certificate. There may also not be a good way to provide the CRL to a system validating a certificate, depending on the network location of the CA relative to the validating system. Is one reachable from the other, and are there firewalls that would preclude the communication? As a result, the Online Certificate Status Protocol (OCSP) can be used to verify a certificate with a CA.

Trusted Third Party

An advantage to using a CA is that there is a central authority that is used not only to store certificates and manage them but also to perform verification of identity. Remember that you provide identification information into the certificate. Nonrepudiation doesn't work if the certificate doesn't verifiably belong to anyone. Just providing a name and even an email address is insufficient to verify someone's identity. It could be spoofed, or it could be coming from an attacker who has access to the

correct email address. The way to address this is to have someone who actually verifies identity. This may come from checking some identification credential that demonstrates that you are who you say you are. In the case of a third-party provider like VeriSign, you may be expected to securely provide a photo identification before they will issue your certificate.

This brings in the idea of a trusted third party. Ideally, you want to always know that the person you are sending encrypted messages to or receiving signed messages from is the person you expect them to be. This works with a CA because of the transitive property. The transitive property says, in this case, that if you trust the CA and the CA trusts that another person is who they say they are, then you, too, believe that person is who they say they are. I trust you, you trust Zoey, therefore I trust Zoey. This assumes that the CA is doing a full validation on the person who has requested the certificate.

We know which CA to trust because every certificate that is generated by a CA is signed by the CA's root certificate. This means the CA adds a digital signature generated by its private key. The system checking the certificate would verify that the signature matches the CA's public key. If that's true, you know the certificate is authentic and the identity associated with the certificate is valid. This means that for every CA you trust, you need to add that CA's root certificate to your certificate cache so the signatures match something.

Ultimately, it's up to the endpoint to validate the certificate. You may have run into cases where there have been certificate errors. This may be a result of a certificate being generated from an untrusted authority, meaning you haven't installed that authority's root certificate. It may also be a result of rogue certificates, meaning someone generated a certificate to look like one thing when in fact it's something else entirely. It may also be a result of a misconfiguration. You can see an example of this in Figure 13.8.

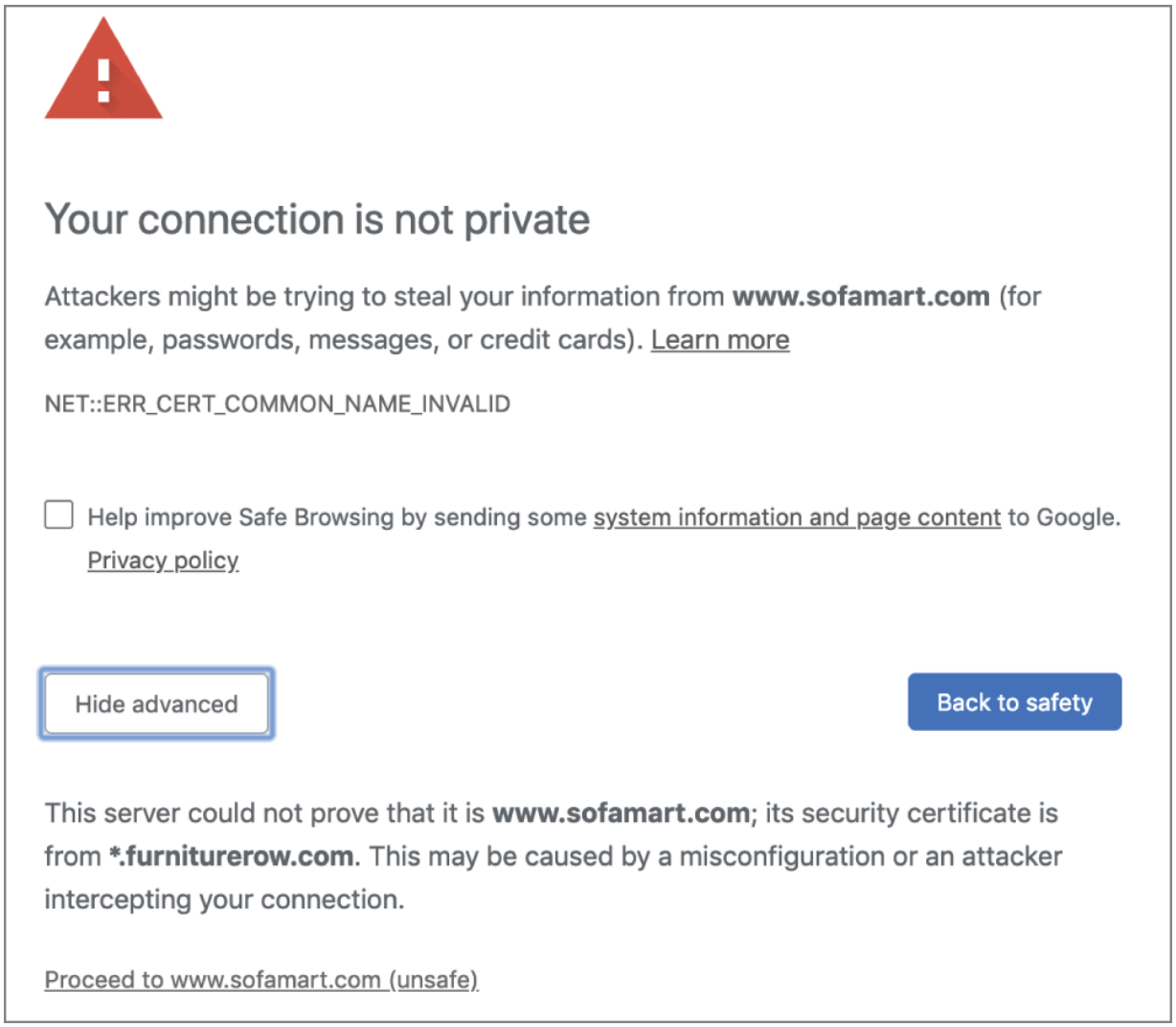

FIGURE 13.8 Certificate error

Google Chrome has generated this error because the name on the certificate doesn't match the name of the server. The hostname of the web server is `www.sofamart.com`. When you register for a certificate, you have to provide the name. This includes the hostname of the server you are installing the certificate to. These values must match. In the case here, the certificate was issued to `*.furniturerow.com`. This demonstrates that even if a certificate comes from a trusted authority, that doesn't mean the server you are connecting to can be trusted because the certificate may have been taken and installed somewhere else. Here, Furniture Row is the company that owns Sofa Mart, so it attempted to save some money by using the same certificate across all servers it uses.

Self-Signed Certificates

Sometimes you want to just encrypt messages between two systems in a lab, for instance. You can create your own certificates. You don't even need to go through the steps of creating your own certificate authority. You can just generate your own certificate and use it. This will result in certificate errors because the certificate won't be signed. It will be a bare certificate. Just as OpenSSL was used by Simple Authority, it can be used on the command line to generate certificates by hand. You would be using the commands Simple Authority is running. Here you can see the OpenSSL commands used to create a certificate.

Certificate Generation

```
 root@quiche:~# openssl req -x509 -newkey rsa:4096 -keyout
key.pem
 -out cert.pem -days 365
 Generating a RSA private key
 ..........................................++++
 ..................................................................................................................................................++++
 writing new private key to 'key.pem'
 Enter PEM pass phrase:
 Verifying - Enter PEM pass phrase:
 -----
 You are about to be asked to enter information that will be
incorporated <![CDATA[into your certificate request.
 What you are about to enter is what is called a Distinguished
Name or a DN.
 There are quite a few fields but you can leave some blank
 For some fields there will be a default value,
 If you enter '.', the field will be left blank.
 -----
 Country Name (2 letter code) [AU]:US
 State or Province Name (full name) [Some-State]:CO
 Locality Name (eg, city) []:Wubble
 Organization Name (eg, company) [Internet Widgits Pty Ltd]:
 Organizational Unit Name (eg, section) []:IT
 Common Name (e.g. server FQDN or YOUR name) []:www.server.com
 Email Address []:wubble@wubble.com
```

I used the `openssl` command to issue a request for an X.509 certificate. That gets done in the first part of the command line. Next, I provided details

about the key that will be embedded into the certificate. It's an RSA key that is 4,096 bits in length. The key gets written out to the file `key.out`, while the certificate itself is written out to `key.pem`. I also need to indicate how long the certificate is good for. You may have run across cases where the certificate is out-of-date, meaning an old certificate is still being used. Your browser may disallow connections to servers with old certificates. You can specify any time period you would like here. What has been requested on the command line is 365 days. You may have noticed when I created my own CA, the root certificate has an expiration of 10 years. You don't want to have to keep updating your root certificate every year because it would lead to a lot of users who had outdated root certificates, meaning every certificate signed by that CA would be invalid. Again, you run into the transitive property at work.

This is the first time you've seen an actual file get created for a key or a certificate. If you have these files, you can use `openssl` to extract information from them. Here you can see what that looks like. Again, I am performing an `x509` function. The parameters are much simpler than those used when generating the certificate. I provide the input file, which is the certificate file generated earlier. Then, I just tell `openssl` to generate text output.

Showing Certificate Details

```
root@quiche:~# openssl x509 -in cert.pem -text
Certificate:
    Data:
       Version: 3 (0x2)
       Serial Number:
5a:50:cd:8a:dc:2d:57:ff:52:13:ca:51:a6:f5:90:7e:71:28:50:9a
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, ST = CO, L = Wubble, O = Internet
Widgits Pty Ltd,
OU = IT, CN = www.server.com, emailAddress = wubble@wubble.com
        Validity
            Not Before: Jan 20 22:13:54 2019 GMT
            Not After : Jan 20 22:13:54 2020 GMT
        Subject: C = US, ST = CO, L = Wubble, O = Internet
Widgits Pty Ltd,
OU = IT, CN = www.server.com, emailAddress = wubble@wubble.com
        Subject Public Key Info:
```

```
Public Key Algorithm: rsaEncryption
    RSA Public-Key: (4096 bit)
```

Here you can see a lot of details about the certificate that weren't provided. First, there is a version number, which indicates the version of the X.509 certificate that is in use. This changes the functionality that is included in the certificate. There is also a serial number, which uniquely identifies the certificate. We also have details about the issuer. In most cases, this is the CA. In this case, the issuer and the subject are the same because there is no authority. It's a self-signed certificate, so the same device it's been generated for has effectively signed it.

There are a lot of details missing from the preceding snippet of text output. For a start, the fingerprints for the key are not included. Additionally, the full text output includes the modulus, which is the product of two large prime numbers. This is a portion of the key. Finally, you will also get a Base64-encoded representation of the certificate. There is a lot of data that is stored in the certificate, and it takes a lot of space to print it all out. It does, though, give you a good sense of what information is captured and is necessary to make certificates work.

Cryptographic Hashing

There is an important element to cryptography that we have been breezing by so far. You may have seen it show up in places earlier in the chapter, such as the self-signed certificate or even the ciphersuites used on a web server. The important part that's been missing is verification of data. You can encrypt data, but as part of that, you should also be making sure that what is sent is what has been received to ensure that nothing has been tampered with. This can be done using a message authentication code (MAC). The MAC here is a fixed-length value that is generated by running the entire message through a cryptographic algorithm. The output is often referred to as a *hash*. It can be used for multiple purposes beyond just being able to verify a message that has been sent.

A common hash function, though starting to be deprecated, is Message Digest 5 (MD5). MD5 is a cryptographic algorithm that takes arbitrary-length input and generates a fixed-length output. Generating an MD5 hash

will yield 32 hexadecimal characters, which is 128 bits. When it comes to cryptographic hashes, it's not a linear function. This means even the change of a single bit will generate a completely different value. You can see this in the following code listing where I take the string `‘123456789’` and get an MD5 hash from it using the `md5sum` command. The second time around, I get a hash from the string `‘223456789'`, which is a change of a single bit. The difference in MD5 output is completely different. One value has no relation to the other in spite of being different by a single bit.

Generating an MD5 Hash

```
root@quiche:~# echo '123456789' | md5sum
b2cfa4183267af678ea06c7407d4d6d8  -
root@quiche:~# echo '223456789' | md5sum
2cad6f3fd5e54b84cfccd3e1ef5aea4d  -
```

One of the expectations of a cryptographic hash is that every set of input will yield a different value. When two different sets of input generate the same output value, it's called a *collision*. These collisions, especially if they can be manufactured, could lead to exploitations. Since these values are used to authenticate messages, if the message digest output could be manipulated through the use of collisions, it could result in messages being altered while still allowing the message digest value to remain the same.

The problem of collisions in hash algorithms relates to a mathematical problem called the Birthday Paradox. This states that in a room of 23 people, there is a 50 percent chance that two people share the same birthday (month and day). However, to get to 100 percent, you have to have 366 people in the room. The graph of the probability has a steep ramp early on but then essentially trails along the top for a while. It's this probability game of being able to generate a collision in hash values that becomes the attack vector.

MD5 is not the only algorithm used for cryptographic hashing. In fact, it has generally been replaced by the Secure Hash Algorithm 1 (SHA-1). It provides a 160-bit output, which is 20 bytes. This is represented as 40 hexadecimal digits. The same is true with SHA-1 as it is with MD5. A single bit difference in two sets of input generates two sets of output that have no relation to one another. You can see that here using the same demonstration from the MD5 hash demonstration.

Hash Values with SHA-1

```
root@quiche:~# echo '123456789' | sha1sum
179c94cf45c6e383baf52621687305204cef16f9  -
root@quiche:~# echo '223456789' | sha1sum
884228b47dd406b235d020315929a8841f178a93  -
```

These hash functions can be used to authenticate messages because the hash is transmitted with the message. If there is any difference in the message, it should be considered not to be authentic. If it isn't considered authentic, it should be discarded because the assumption is that it has been tampered with. Again, this is why not being able to manipulate the message digest is essential. Once an attacker can manipulate the message and still have the digest come out right, they can change encrypted messages that are believed to be private.

SHA-1 is not the end of the line. Currently, there is SHA-2, which supports four different digest sizes, or bit lengths, that the algorithm can generate. These are 224, 256, 384, and 512 bits. There is also SHA-3. It supports the same lengths as SHA-2 but differs internally from SHA-1 and SHA-2. Since SHA-2 has digest lengths up to 512 bits, there isn't really any reason not to use it rather than SHA-1 and certainly MD5. It has better protection against collisions and doesn't take substantially more time to calculate even with the larger key space.

In addition to providing message authentication, cryptographic hashes are used to verify data as being correct. This is done often with downloaded files from websites. When the file is downloaded, a hash can be computed on what was downloaded and compared to what it should be. In this case, it doesn't protect against authenticity of the file, meaning there isn't much of a

chance of tampering during the download. Instead, you are comparing against corruption during the download process. This is not to say that there aren't cases where the underlying file may have been swapped, and you should make sure that what is expected to be available through a website for download is actually the file that is available for download. This means when a file is changed in the file system, the website needs to be updated as well.

Cryptographic hashes are commonly used for file integrity systems like Tripwire. Essential files in the file system have a hash computed and stored. The software then goes through periodically to compare what is on the file system to what is in the known good database. If there is a change, it is flagged as a possibility of system compromise. Host-based intrusion detection systems will often use elements like this to be able to detect when an attacker gains access to a system.

PGP and S/MIME

Public key encryption is commonly used in multiple situations. However, CAs are not always used. Not everyone appreciates a centralized management approach, where organizations are trusted. In the 1990s, Phil Zimmermann developed another way of managing certificates that does not use a CA for centralized verification. Instead, Pretty Good Privacy (PGP) uses a “web of trust” to perform verification. The idea is that keys are all uploaded to a web server. Someone who knows the person who has uploaded their key will sign that key as a demonstration that they know the person and are willing to say that key really belongs to the user it purports to belong to.

Say Zoey's email address is zoey@wubble.com and she uploads her PGP key. This is based on the creation of an X.509 certificate with a public key. It is the public key that is being stored in the public web server. I happen to know Zoey. I know her email address, and she tells me what the fingerprint of her public key is. I go to the PGP key site and sign her key with my signature. Anyone who knows me but doesn't know Zoey can be assured that Zoey's key is legitimate. This is again an example of the transitive property. You trust me, and I trust that Zoey's key is legitimate, so you can

trust that Zoey's key is legitimate, and you can send encrypted messages to her using her public key.

You end up managing all the public keys you want to communicate with by storing them in a keyring. This keyring would typically be signed with your key and also stored with a message authentication code so you can be sure it hasn't been tampered with. As you have likely seen often by now, when you are talking about encryption and privacy, ensuring the validity of every aspect of the process and protecting the keys are important. A downside of PGP is that keys are managed by users. This means that you have to keep track of your key and keep it with you. Theoretically, I would have a 30-year-old PGP key. Unfortunately, I ended up having to rebuild systems or had them mistakenly deleted.

This means I had to keep re-creating keys. This can lead to confusion, as you can see in Figure 13.9, where there are multiple keys for the same email address. How would you know which one you should be using to encrypt a message? It puts all of the management into the hands of the individual user. The user has to know who they want to send messages to and which key to use that would be correct for that user.

Search results for 'ric messier'

Type	bits/keyID	Date	User ID
pub	2048R/77BC3732	2013-05-13	Ric Messier <kilroy@WasHere.COM>
pub	1024D/507D2485	2000-03-28	Ric Messier <rmessier@bbnplanet.com>
pub	1024D/A6CCD851	2000-01-16	Ric Messier <kilroy@WasHere.COM>
pub	1024D/C08CFEE1	1998-10-09	Ric Messier <ric@segNET.COM>
pub	1024D/BAD133F1	1998-08-27	Ric Messier <kilroy@WasHere.COM>

FIGURE 13.9 List of PGP keys

PGP doesn't work well for certificates for servers. Instead, it's used to send messages like email from one user to another. PGP can be used to encrypt an email message that is sent from one user to another, as long as they are running PGP software. PGP is not the only solution for email encryption, though.

Secure/Multipurpose Internet Mail Extensions (S/MIME) is another protocol for sending encrypted email messages. This is a standard that is generally implemented in email clients, meaning there is no need for third-party software. S/MIME also uses X.509 certificates from certificate authorities. These certificates may commonly be installed inside a Windows Active Directory. In a fully Microsoft environment, users within an enterprise can send encrypted messages to one another and public keys can be retrieved from the Active Directory server. This means there is no need to have other root CA keys in the system or require any other methods to get the public key onto the client system.

Disk and File Encryption

From a data security perspective, there are three types of data. The first is data in motion, which we covered already by talking about the use of S/MIME and hybrid cryptosystems. These are techniques to encrypt data that is being transmitted from one location to another, commonly over a network. The data is moving from a sender to a recipient, so it is said to be in motion. The second type of data is data in use. This is data that is being acted on by an application, so it is commonly in memory. This type of data is harder to address from a security perspective because it generally can't be encrypted while it is being acted upon. Instead, there are periods of time where the data is in the clear, which is problematic. Developers may use techniques like data masking, which is a way of keeping the data usable but keeping it segmented so all of the data isn't viewable at one time.

The final type of data is data at rest. This is data that is stored to something like a disk or a tape. Even data that has been backed up needs to be protected, after all, so there should be consideration for encrypting backups. However, the most common approach to data at rest is either file-level or disk-level encryption. File-level encryption can be handled by many utilities. As one example, on a macOS system, Preview can encrypt files. Figure 13.10 shows the dialog box from exporting a PDF with the Encrypt File option checked. To encrypt the file, you would need a password to serve as the encryption key.

Other utilities like PGP, previously discussed, can also encrypt files as well as file systems. When it comes to encrypting file systems, a good approach

is generally to make use of the encryption built into the operating system. All of the major operating systems you will run across have encryption built in. When it comes to this level of encryption, it's handled so low in the operating system and uses what we call full disk encryption that you need to provide authentication before the operating system even boots up. This would unlock the key. Many hardware systems include something called a *trusted platform module* (TPM). This is a cryptographic processor on a chip, which can perform tasks like storing keys that would decrypt the file systems. The key would be strongly protected, and only the right authentication would be able to unlock the key. The TPM is something that may be found on systems that are meant to run Windows. Apple has its own security module it has developed that performs similar functions.

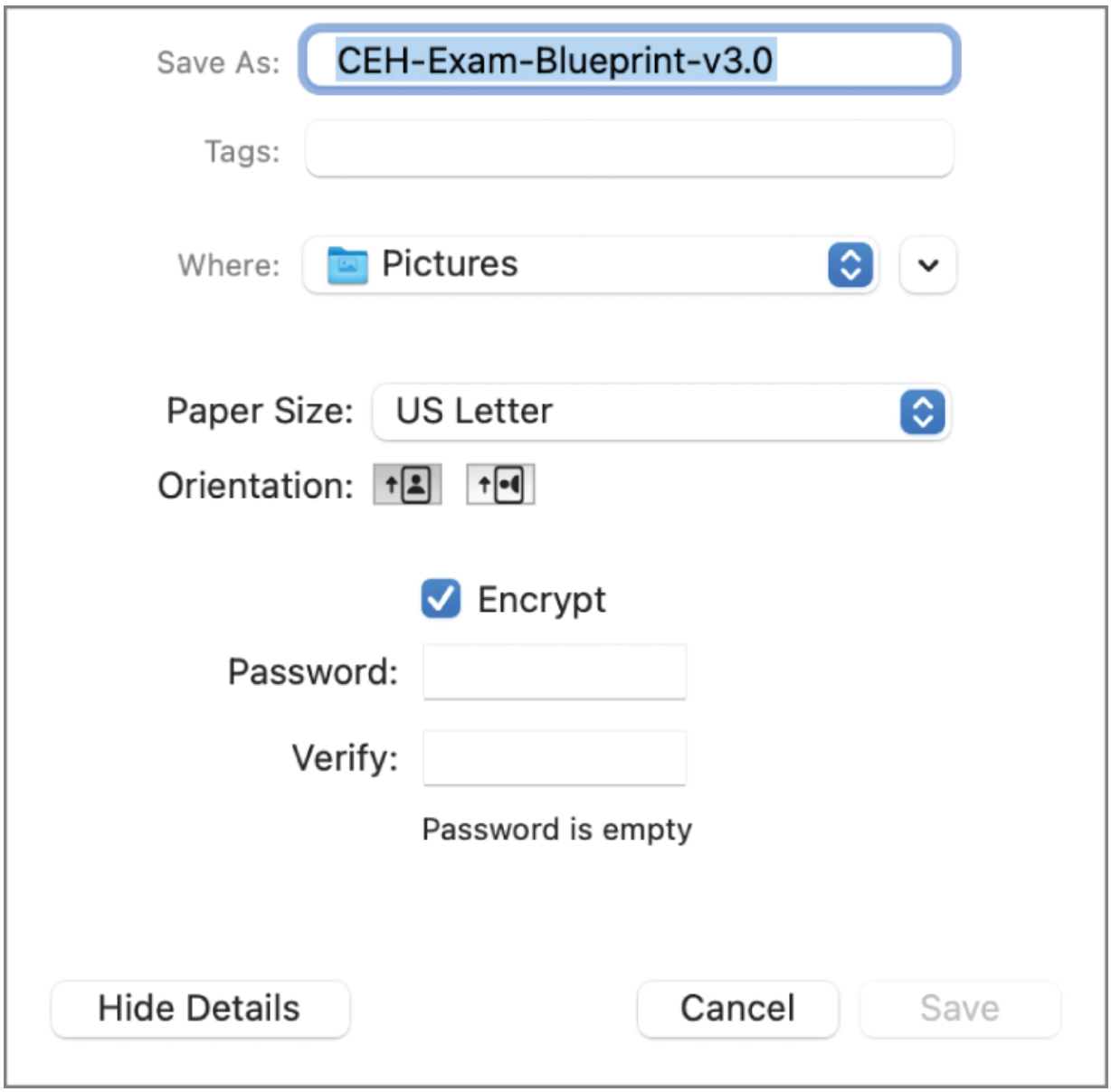

FIGURE 13.10 Encrypting a PDF on macOS

Apple uses a full disk encryption utility called FileVault. It's built into the operating system and uses AES to encrypt volumes. You access FileVault from the System Preferences application under Privacy & Security. Figure 13.11 shows the System Settings tool where FileVault has just been enabled and an administrator's password is required to turn it on. Once you enable FileVault, you will have to authenticate to the system before it will boot. Additionally, when you enable FileVault, you will get a backup decryption key. It's essential to store this somewhere other than on the volume that has been encrypted. If you manage to forget the password for the account that

unlocks the drive, having the key on that drive won't help much. Just as with any decryption key, though, you have to protect it. If the key is accessible to anyone, it can be used to decrypt the drive, which would essentially negate the reason for encrypting to begin with.

Encrypted Drives

There is a problem with encrypted drives, however. There is a limited problem set they have any use for. An encrypted drive is only good to protect against something called *dead box access*. If I have the drive, encryption will prevent me from reading it unless I have the key. If I manage to get access to the system remotely in some way, the drive will appear as though it were decrypted, just as it does when you are using your system. Any attacker who can get access to a system with an encrypted drive will appear to the system as though they were an authenticated user, so the drive will be decrypted when they attempt to access it.

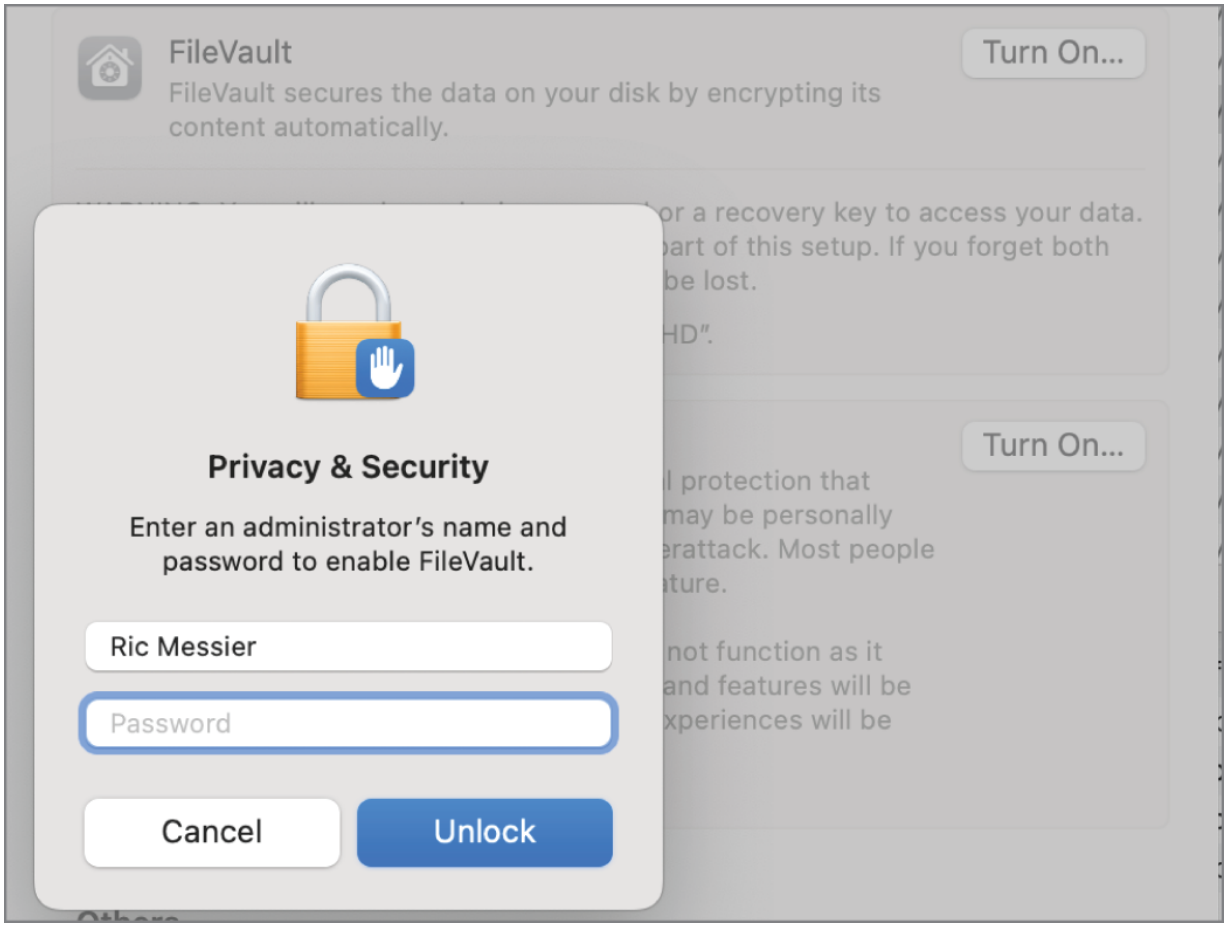

FIGURE 13.11 FileVault encryption

Windows also has encryption software built into it. This is called BitLocker. Similar to macOS, you can get to it by going to Updates & Security in Settings on a Windows system. Figure 13.12 shows the Control Panel page with the BitLocker settings on it. When you enable BitLocker, you will be asked about storing the decryption key. It will be stored in the TPM chip if it's available. However, that's the primary key. You still need to deal with the backup key. In a Windows network, you probably have an Active Directory server. The backup key can be escrowed within Active Directory. This provides a central point where the key can be retrieved. In the case of a business, after an employee leaves, taking the password that can unlock the drive, the key escrowed within Active Directory allows the business to retrieve essential information, if needed. There is a way to retrieve the key after it has been taken away when the employee leaves.

FIGURE 13.12 BitLocker control panel

Let's not forget Linux. There have been different iterations of encryption on Linux systems. The current one is called `dm-crypt`, and it resides within the Linux kernel. This is a slightly different process than just opening options and enabling. However, `dm-crypt` also allows us to fairly easily encrypt external volumes. The following is the process you would follow to encrypt an external volume on a Linux system. It starts with a volume created on a USB drive. We first have to add the Linux Unified Key Setup (LUKS) header to the drive. This will require a password that will protect the key and, by extension, the data on the drive. Once the encrypted volume has been created, it has to be opened before you can do anything with it.

Encrypting Disk in Linux

```
kilroy@hodgepodge $ sudo cryptsetup luksFormat /dev/sdf1

WARNING!
========
This will overwrite data on /dev/sdf1 irrevocably.

Are you sure? (Type uppercase yes): YES
Enter passphrase for /dev/sdf1:
Verify passphrase:
kilroy@hodgepodge $ sudo cryptsetup open /dev/sdf1
encryptedbackup
Enter passphrase for /dev/sdf1:
 kilroy@hodgepodge $ sudo mkfs.ext4 /dev/mapper/encryptedbackup
mke2fs 1.44.1 (24-Mar-2018)
```

```
Creating file system with 3906555 4k blocks and 977280 inodes
File system UUID: 21b73264-143f-40fa-8ca5-156363783a8a
Superblock backups stored on blocks:
         32768, 98304, 163840, 229376, 294912, 819200, 884736,
1605632, 2654208

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and file system accounting information:
done
```

Keep in mind that as soon as you have added the encryption header to the drive, any data that may have been on the drive previously will be gone. Once you've opened the drive using `cryptsetup open`, you will need to create a file system so you can use the drive. As this particular drive is a USB drive, it can be moved from one system to another as long as the password is known. There may be drivers that can be installed on a Windows system, which may allow the drive to be opened there.

Today, there are a lot of ways to encrypt data, either encrypting the full disk or encrypting individual volumes or individual files. Keep in mind, though, that encryption is only as good as the key, so if the key isn't protected well, or someone can get access to a system that is in an authenticated state (someone is logged in and the system isn't locked), the data is effectively unencrypted.

Summary

Encryption is an important concept because privacy is so important. This is especially the case when attackers are looking for any advantage they can get. If they can intercept messages that are not encrypted, they may be able to make use of the contents of the message. Users will sometimes make the mistake of believing that messages sent to other users within an enterprise are safe because they remain inside the enterprise. These messages are not safe because they can be intercepted and used. The same can be true of disk-based encryption. You can't assume that a disk that has been encrypted is safe. Once someone has authenticated as a legitimate user, the disk is unencrypted. This means if an attacker can gain authenticated access, even

by introducing malware that is run as the primary user, the disk is wide open to the attacker.

There are two types of encryption when you think about the end result. The first is substitution, where one character is substituted for another character. This is common with encryption schemes like a rotation cipher and the Vigenère cipher. The second type is a transformation cipher. This is where the unencrypted message, or plain text, is not replaced one character at a time but the entire message is transformed, generally through a mathematical process. This transformation may be done with fixed-length chunks of the message, which is a block cipher. It may also be done byte by byte, which is how a stream cipher works. With a block cipher, the data size is expected to be a multiple of the block size. The final block may need to be padded to get it to the right size.

Key management is essential. An important element of that can be key creation. You could use pre-shared keys, which could be learned or intercepted while they are being shared. If you don't use a pre-shared key, the key could be derived. This may be done using the Diffie-Hellman key exchange protocol. Using a common starting point, both parties in the process add a value and pass it to the other party. Once the value has been added to the shared key, you end up with both sides having the common value plus the random value from side A plus the random value from side B. Both sides have the same key and can begin sending encrypted messages.

This process could be used for symmetric keys, where the same key is used for both encryption and decryption. The Advanced Encryption Standard (AES) is a common symmetric key encryption algorithm. AES supports 128, 192, and 256 bits. You might also use an asymmetric key algorithm where different keys are used for encryption and decryption. This is sometimes referred to as *public key cryptography*. A common approach is to use a hybrid cryptosystem where public key cryptography is used to share a session key, which is a symmetric key used to encrypt messages within the session.

Certificates, defined by X.509, which is a subset of the X.500 digital directory standard, are used to store public key information. This includes information about the identity of the certificate holder so verification of the

certificate can happen. Certificates may be managed using a CA, which is a trusted third party that verifies the identity of the certificate holder. A CA is not the only way to verify identity, though. PGP uses a web of trust model, where individual users validate identity by signing the public keys of people they know.

A MAC is used to ensure that messages haven't been altered. This is generally a cryptographic hash, which is an algorithm that generates a fixed-length digest value from variable-length data. This can be used not only for message authentication but also for verifying that files have not been tampered with or corrupted.

Full disk encryption is a common technique on systems today. Windows systems will typically use BitLocker, though there are third-party software solutions that will perform the same function. On macOS systems, you would use FileVault, and Linux uses `dm-crypt`, which is built into the kernel but still requires utilities to be installed on the system to handle setting up volume encryption and then opening encrypted volumes. As with any encryption, key management is essential, and encrypted files or file systems are not protections against everything. Encrypted files may be a good way of losing data if the key or password is lost.

Review Questions

You can find the answers in the appendix.

1. With a rotation of 4, what does erwaiv decrypt to?

    A. *waive*

    B. wave

    C. answer

    D. decrypt

2. What do you call a message before it is encrypted?

    A. Text

    B. Plain text

    C. Bare words

D. Bare text

3. What does PGP use to verify identity?

A. Central authority

B. Web of users

C. Web of trust

D. Central trust authority

4. What principle is used to demonstrate that a signed message came from the owner of the key that signed it?

A. Nonrepudiation

B. Nonverifiability

C. Integrity

D. Authority

5. What is Diffie-Hellman used for?

A. Key management

B. Key isolation

C. Key exchange

D. Key revocation

6. How did 3DES improve on DES?

A. Made the key longer

B. Used two keys

C. Changed algorithms

D. Used three keys

7. What improvement does elliptic curve cryptography make?

A. Smaller keys improve speed

B. Algorithm is more complex

C. Doesn't use factoring, which is better

D. Longer keys

8. What is it called when two different data sets yield the same cryptographic hash?

   A. Paradox

   B. Collision

   C. Crash

   D. Unrealistic

9. Which of the following terms can be used in a description of asymmetric key encryption?

   A. Defined key

   B. Multifactor

   C. Public key

   D. Single factor

10. If Alice were to send an email to Bob, what key would she use to encrypt the message?

    A. PGP key

    B. Private key

    C. Public key

    D. Symmetric key

11. What property allows you to trust someone trusted by a certificate authority you trust?

    A. Commutative property

    B. Associative property

    C. Communicative property

    D. Transitive property

12. Why is symmetric key encryption typically used over asymmetric key encryption?

    A. It's faster.

B. It's more secure.

C. It's easier to implement.

D. It isn't encumbered with patents.

13. What is it called when both symmetric and asymmetric keys are used?

A. Fast cryptosystem

B. Hybrid cryptosystem

C. Super-symmetric cryptosystem

D. Dual key cryptosystem

14. What is MD5 or SHA-1 commonly used for in cryptography?

A. Media access control (MAC)

B. Machine authentication code (MAC)

C. Message access code (MAC)

D. Message authentication code (MAC)

15. What type of encryption does PGP use?

A. Null key

B. Asymmetric key

C. Trusted key

D. Web key

16. What tool would you use to identify ciphersuites in use on a web server?

A. `Hydra`

B. `sslscan`

C. `tlsscan`

D. `cipherscan`

17. How does AES protect against related-key attacks?

A. Longer key lengths

B. Better initialization vectors

C. Implementation doesn't allow related keys

D. Upgrading to AES-2

18. What is one advantage of using a certificate authority?

A. Trusted third party doing validation

B. A certificate authority is faster

C. Stronger keys are offered

D. They support more cipher suites

19. How does a certificate authority keep a list of valid certificates up-to-date?

A. Periodic CA update

B. Hashing the list

C. Re-validating identities

D. Certificate revocation lists

20. What security property suggests that an email signed by an individual's key must have come from that person?

A. Confidentiality

B. Integrity

C. Availability

D. Nonrepudiation