Chapter 22

Cryptographic Gotchas

This chapter describes a number of cryptographic attacks (gotchas) and explains how you can foil them.

Replay Attack

Suppose Alice sends her clients the daily interest rate and a buy/sell recommendation, signed with her private key. All her clients have trusted copies of her public key. On Monday, Alice sends 6.5% don't buy yet. Because this message is only signed and not confidential, BlackHat decrypts it. On Tuesday, the Fed, in uncharacteristic mania, drops rates to 5.5%; Alice sends 5.5% buy now. Because BlackHat wants to stall Alice's clients, he intercepts Alice's Tuesday message (5.5%) and substitutes Alice's Monday message (6.5%). It's an authentic Alice message; BlackHat didn't alter it. It's just old. This attack, appropriately called a *replay* attack, is one of the more easily prevented attacks.

Lesson: Timestamp or number messages

To prevent a replay attack, Alice can timestamp or number her messages. All the real-world systems we've discussed offer at least one of these options. Interestingly, although IPsec requires the sender to number messages, the receiver is not required to use the sender's message numbering.

Man-in-the-Middle Attack

Although public keys need not be concealed (secret), this doesn't mean that public keys can simply be sent (or stored) without any protection. For example, suppose Alice e-mails Bob her public key; then BlackHat intercepts it and substitutes his own (BlackHat's) public key. BlackHat can now read all confidential messages Bob sends to Alice and even masquerade as Alice to Bob. This is called a man-in-the-middle attack.

Lesson: A trusted public key means it's been validated and protected.

In Figure 22-1, BlackHat intercepts Alice's public key (23, 69, 14, …) and substitutes his own (99, 98, 97 …). Bob uses BlackHat's public key (thinking it is Alice's) to encrypt messages for Alice. To complete his subterfuge, BlackHat, after decrypting and reading Bob's message, encrypts it using Alice's public key and sends it to her (see Figure 22-2). Similarly, BlackHat can forge Alice's signature to Bob—again, because Bob believes he has Alice's genuine public key. Is this too much work for BlackHat? That depends on how much he can gain from it, doesn't it?

Bob can verify Alice's public key using digital certificates (see Chapters 15 through 17). But recall that digital certificates also rely on an initial trusted public key.

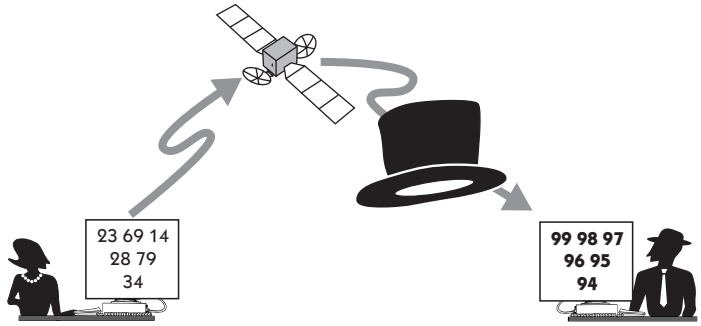

Figure 22-1 Black Hat substitutes his own public key for Alice's. Bob doesn't know.

1. Bob encrypts a message to Alice, unknowingly using BlackHat's public key.
2. BlackHat intercepts and decrypts Bob's message with his (BlackHat's) private key.
3. BlackHat encrypts Bob's message with Alice's public key and sends it on to Alice.

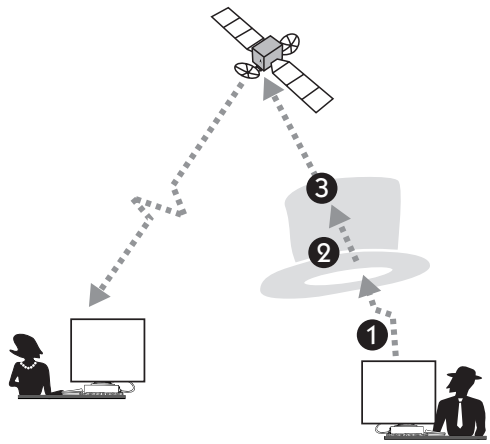

Figure 22-2 BlackHat completes the deception.

Finding Your Keys in Memory

Here's an attack that's so obvious that, after you see it, you might wonder why you didn't figure it out yourself.

Figure 22-3 shows a snapshot of RAM bits; white is 0, and gray is 1. Can you find the simplistic 26-bit secret key? Here's how. Recall that cryptographic keys are made to have as much randomness as possible; that makes them difficult to predict. It also means that a cryptographic key should contain approximately the same amount of 0's and 1's—in our RAM picture, the same number of gray and white boxes.

Random keys don't look like normal data. That makes them easy to find in memory.

The secret key begins four rows down on the left-hand side; about 13 white boxes are alternated with 13 gray boxes. Of course, real secret keys don't alternate every 0 and 1 bit; that isn't a random pattern. But that's not the point. The point is that a random key has statistical properties that "normal" data does not. That gives the cryptanalyst something to look for in RAM: randomness. This means that although random keys are hard to predict, their randomness makes them easy to find.

To protect against this attack, a cryptographic software designer can split the random key into smaller chunks so that its randomness doesn't provide such a big target.

This attack, discovered by Adi Shamir, the *S* in RSA, also illustrates a cryptographic tenet: You can't say that a system is totally secure because you never know where an attack will come from.

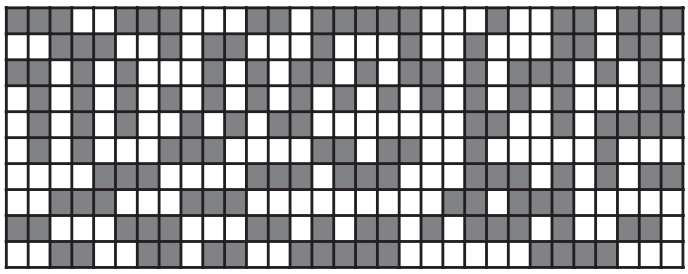

Figure 22-3 Computer memory with a secret key.

Does Confidentiality Imply Integrity?

There's a widely held, but mistaken, notion that secret key encryption (confidentiality) implies that a message can't be altered in transit without detection. The mistaken argument goes like this: The secret key is shared only between Alice and Bob, so BlackHat can't change the encrypted message because

anything he substitutes will decrypt to gibberish (see Figure 22-4). Let's disprove this notion with two examples.

Example 1: Substituting a Forged Key

Lesson: Always include a signed digest or a MAC in digital messages.

What if the ciphertext in Figure 22-4 contained a secret or public key, that is, something random? Although BlackHat can't decrypt the ciphertext, he can substitute some other random ciphertext. When Bob decrypts BlackHat's forged ciphertext, he'll have no way to detect BlackHat's forgery. BlackHat has successfully and covertly disrupted future communications between Alice and Bob. And although BlackHat can't read Alice's communications to Bob, neither can Bob!

Figure 22-4 Alice sends Bob a secret key encrypted message.

Example 2: Cut-and-Paste Attack

Secret key encrypted messages usually break the plaintext into approximately 8-character to 16-character blocks (chunks) and encrypt each individual block separately (see Chapter 5).[1]

Unfortunately, if there are enough blocks of ciphertext, BlackHat can insert fraudulent blocks of ciphertext, as shown in Figure 22-5. For example, encrypted blocks 1–9 may be genuine, but blocks x, y, and z are a forgery inserted by BlackHat.

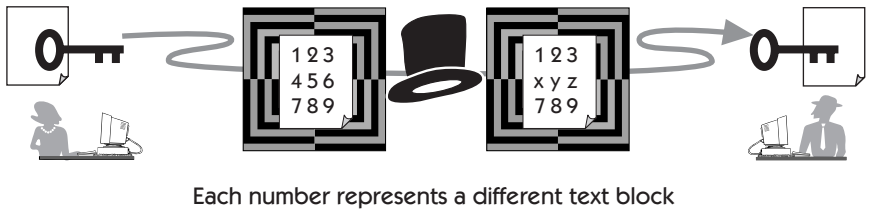

Figure 22-5 BlackHat inserts forged packets into a genuine message that Bob thinks are genuine.

1. Disregarding stream ciphers.

This attack is called a cut-and-paste attack. The details are beyond the scope of this book, but you'll find references in the Bibliography. Note that sending either a MAC or a signed message digest will detect this attack.

Public Key as a Cryptanalysis Tool

When Bob encrypts and sends a message to Alice using her public key, it's supposed to be unreadable by BlackHat. But it isn't always unreadable unless you follow some guidelines.

Example 1: The Chosen Plaintext Attack

Suppose Alice wants Bob's opinion on how much to bid for a shopping center; it is worth between $8,000,000 and $12,000,000. As shown in Figure 22-6, Bob encrypts his opinion ($10,987,654 encrypts to 12341234) and sends it to Alice. How can BlackHat find out what Bob encrypted?

Let's review what BlackHat knows. He knows that Bob's opinion is between $8,000,000 and $12,000,000; the ciphertext Bob sent Alice (12341234); and Alice's public key. BlackHat uses his copy of Alice's public key to encrypt every number between 8,000,000 and 12,000,000. One number, 10,987,654, encrypts to 12341234, the same ciphertext sent by Bob (see Figure 22-7). BlackHat knows that this is the opinion Bob sent Alice.

Definition: chosen plaintext attack

In cryptographic terminology, BlackHat has successfully mounted a *chosen plaintext* attack. BlackHat chooses a set of possible plaintexts to encrypt with Alice's public key; one of the chosen plaintexts encrypts to the ciphertext sent by Bob.

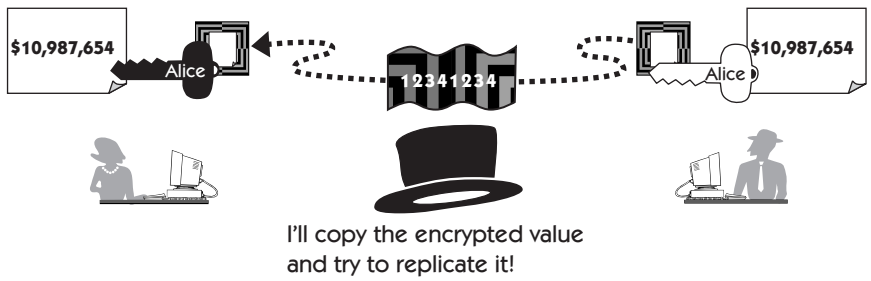

Figure 22-6 Bob sends Alice his encrypted opinion, and BlackHat copies the ciphertext.

Definition: padding

Alice and Bob can prevent a chosen plaintext attack by adding random characters, called *padding*, to Bob's plaintext. This approach effectively increases the number of possible plaintexts BlackHat must encrypt. Figure 22-8 adds five characters of padding. Now BlackHat must encrypt 100,000 variations of 8,000,000 before trying 8,000,001.

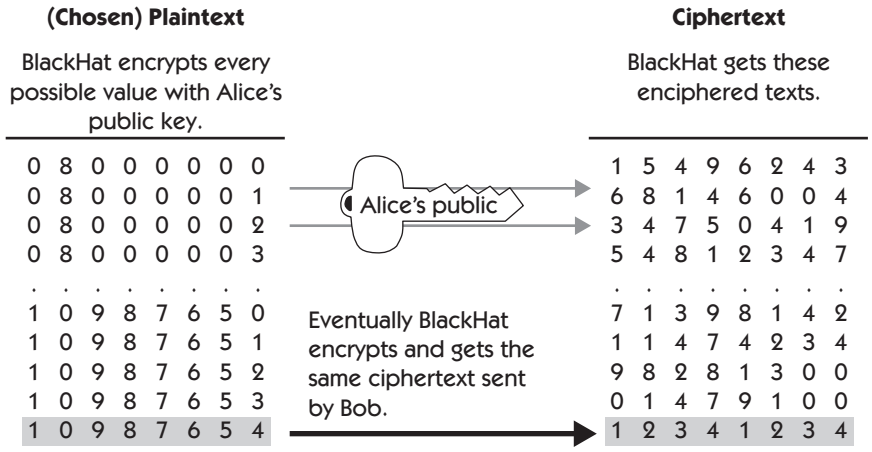

Figure 22-7 BlackHat mounts a chosen plaintext attack.

Additional padding makes BlackHat try 100,000 variations of 8,000,000 before trying 8,000,001

Padding					Bob's Possible Plaintext						
0	0	0	0	0	8	0	0	0	0	0	0
0	0	0	0	1	8	0	0	0	0	0	0
0	0	0	0	2	8	0	0	0	0	0	0
.	.	.	.	.	.	.	.	.	.	.	.
9	9	9	9	7	8	0	0	0	0	0	0
9	9	9	9	8	8	0	0	0	0	0	0
9	9	9	9	9	8	0	0	0	0	0	0
0	0	0	0	0	8	0	0	0	0	0	1
0	0	0	0	1	8	0	0	0	0	0	1
0	0	0	0	2	8	0	0	0	0	0	1
0	0	0	0	3	8	0	0	0	0	0	1

Figure 22-8 Adding padding. BlackHat must try many more than 4,000,000 chosen plaintexts.

Public Key Cryptographic Standards

RSA Securities publishes suggested standards to prevent this and other attacks against public key cryptography. The standards, appropriately called Public Key Cryptographic Standards (PKCS), are freely available from the RSA Web site, www.rsa.com. This particular attack is addressed in PKCS #1, RSA Encryption Standard, which includes a recommended padding scheme.

Example 2: The Bleichenbacher Attack

Unfortunately, as we have mentioned, nothing is provably secure.[2] Even after RSA published PKCS #1, a cryptanalyst, Daniel Bleichenbacher, discovered another attack specifically against the padding scheme.

Lesson: Use only well-trusted and tested cryptographic systems.

Although it was previously thought impossible, Bleichenbacher showed how to figure out the entire encrypted message one bit at a time. The attack requires Alice (the private key holder) to respond to about 1,000,000 of BlackHat's probing messages, but that's not infeasible in automated systems. To counter this attack, called a Bleichenbacher attack, RSA revised PKCS #1 padding. The attack is beyond the scope of this book; see the Bibliography for a reference to additional papers.

BlackHat Uses Bob's RSA Private Key

Bob keeps his RSA private key securely locked up (see Chapter 23) so that no one else can ever use it. Well, here's a way that BlackHat, with a little help from an automated message response system, can trick Bob. BlackHat tricks Bob into using his (Bob's) RSA private key to decrypt a confidential message Alice encrypted with Bob's public key. Watch closely as public/private keys are juggled and confidentiality is breached with an ingenious BlackHat cryptographic maneuver.

As shown in Figure 22-9 (upper left), Alice sends Bob a message. She signs the message (encrypting it with her black private key) and then encrypts with Bob's white public key. In the upper right, Bob receives the message, decrypts it with his private key, and verifies Alice's signature with her public key. In the bottom right, Bob returns to Alice the same plaintext Alice sent him along with

2. Perhaps one of the more tongue-in-cheek rumors is an NSA suggestion for protecting valuable secrets. Grind the hard disk into small pieces, lock the pieces in an expensive safe, dump the safe in the middle of the ocean, and protect the site with highly paid military guards.

a return receipt, proving to Alice that he received her message; that is, he signs with his private key and encrypts with Alice's public key.

Bob's automated message response system will do this for any signed, encrypted messages he receives. Figure 22-10 shows Bob's system sending BlackHat a similar return receipt. Don't look for any BlackHat tricks in Figure 22-10. It's exactly what Bob did for Alice.

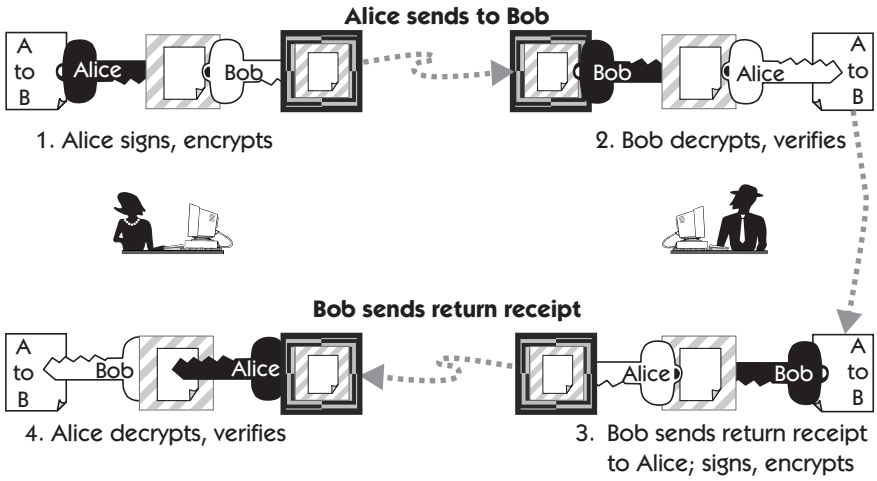

Figure 22-9 Alice and Bob exchange signed, encrypted messages. Follow arrows as Alice signs plaintext and then encrypts, and so on.

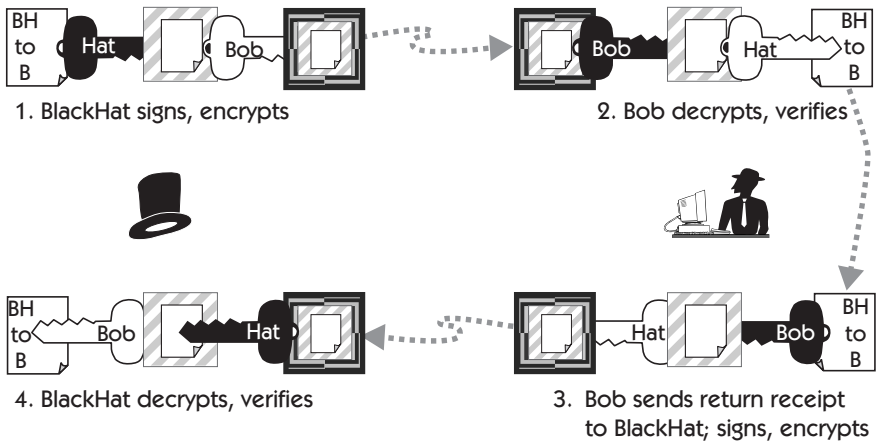

Figure 22-10 Bob's automated response to BlackHat.

It's easier to visualize RSA encryption/decryption (and sign/verify) canceling each other out if it's shown in another way. Figure 22-11 is read from right to left. Alice signs the plaintext message and encrypts it with Bob's public key. Bob decrypts with his private key and verifies Alice's signature. It's exactly identical to Alice and Bob's message exchange shown at the top in Figure 22-9.

Figure 22-11 shows that Bob's RSA private key/public keys cancel each other out. When they are next to each other it's like multiplying by 1. Then Alice's private/public keys cancel each other out, too.

Figure 22-12 adds Bob's return receipt to Figure 22-11. After Bob recovers the plaintext, his automated system makes Alice a return receipt by signing and encrypting. Now we're ready for BlackHat's attack.

In Figure 22-13 BlackHat replays the original signed, encrypted message Alice sent to Bob; it's labeled "BlackHat resends."

Because Bob's automated response system believes that the message is a genuine message from BlackHat, Bob's system dutifully decrypts it with Bob's private key and verifies it with BlackHat's public key. Bob's private key cancels out the public key encryption. Of course, BlackHat's public key won't cancel out Alice's private key signature, and the automated response system won't recover meaningful text. BlackHat hopes that the automated response system simply makes a return receipt automatically, signs with Bob's private key, and encrypts with BlackHat's public key.

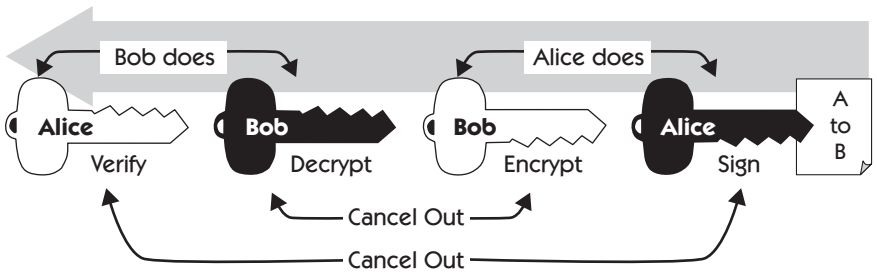

Figure 22-11 Alice sends a signed, encrypted message. Bob decrypts and verifies. This is identical to Figure 22-9 steps 1 and 2.

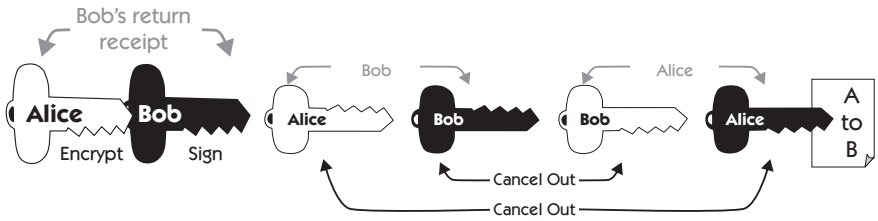

Figure 22-12 Bob's return receipt is added here. This is identical to Figure 22-9 steps 1, 2, and 3.

Figure 22-14 shows how BlackHat figures out the original signed, encrypted message Alice sent Bob. BlackHat has fooled Bob into using Bob's private key to cancel out Alice's use of Bob's public key. See Bob's circled private/public keys.

Although Bob applies three more cryptographic keys to the message—verify with BlackHat's public key, sign with Bob's private key, and encrypt with BlackHat's public key—BlackHat can cancel out each one. Obviously, BlackHat can cancel out any encryption or signing done with BlackHat's public or private key. The only other canceling BlackHat must do is to cancel out signing (private key encryption); he does that with openly available public keys.

Before you start worrying, let's consider the implications of this attack. First, it works only with a cryptographic method whose public *and* private keys are used for encryption *and* decryption—that is, RSA (see Chapter 12). Second, BlackHat figured out only one message encrypted with Bob's public key; he did not figure out Bob's private key.

Lesson: Never sign (private key encrypt) unknown messages.

It's easy to protect against this attack; here are three ways. If you use RSA for confidentiality, use a different method (e.g., DSA) for signing. Or don't use the same pair of RSA public/private keys for both confidentiality and signing; instead, use one RSA key pair for signing and a second one for everything else.

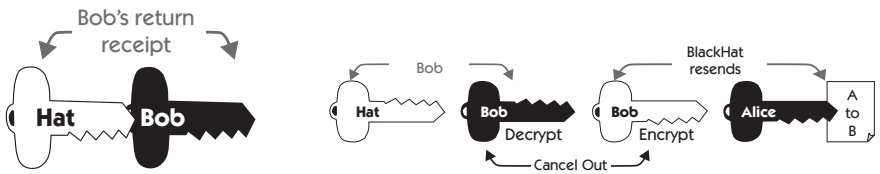

Figure 22-13 Bob's automated response system sends BlackHat a response.

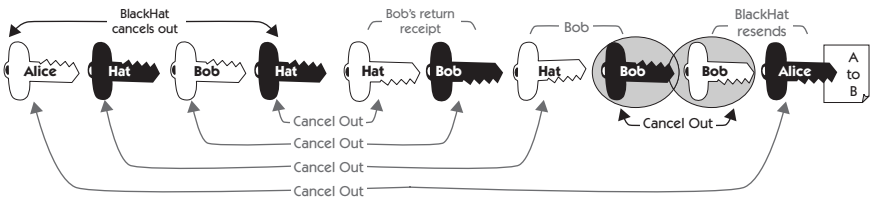

Figure 22-14 BlackHat figures out the encrypted message sent by Alice.

Or better yet, always use a well-designed cryptographic protocol, such as SSL v3, that never signs exactly the message sent to it. Recall, from Chapter 14, a basic tenet of cryptographic protocol design: *Never sign the exact data sent to you*.

Review

Attacks against cryptographic systems are as creative as the cryptographic systems themselves. There are lessons to be learned from looking at each attack.

Replay attack: Messages should contain a timestamp or some other way to identify them as new or old.

Man-in-the-middle attack: Trusted public keys must be validated and protected.

Finding keys in memory: If keys can be identified, they can be attacked when they are outside cryptographic protections.

Confidentiality does not imply integrity: Secret key encryption does not prevent BlackHat from altering a message.

Public and private keys must be used carefully. Adhere to standards, use separate methods and/or keys for confidentiality and signing, and never use your private key to sign the exact message sent to you. And never sign unknown messages.