CHAPTER 13

PUBLIC KEYS AND OFF-LINE AUTHENTICATION

Three can keep a secret if two are dead.

— Ben Franklin, *Poor Richard's Almanac*

IN THIS CHAPTER

Public key cryptography uses clever mathematics to simplify and even eliminate several problems we have with shared base secrets. This gives us the tools we need to implement off-line authentication.

- Public key cryptography and its use in authentication
- RSA public keys and attacks on them
- The U.S. Digital Signature Standard
- Challenge response authentication with public keys
- Secure Sockets Layer
- Public keys for Kerberos preauthentication
- Public keys and biometrics

13.1 PUBLIC KEY CRYPTOGRAPHY

Ben Franklin's famous quote captures the essence of the problem with shared secret keys: we can't prevent the secret from spreading further through mistakes or through transitive trust. This can pose real problems when we use cryptography to authenticate important documents, like electronic checks. On the other hand, if we can find a mechanism that lets us safely sign checks, we can adapt it to authenticate people.

Consider the following situation: Tim Dore has written an electronic check to John Doe, payable by his bank ("I owe John $100").

In a perfect world, Tim gives this electronic check to John, and John forwards it to the bank. Both John and the bank must be able to authenticate the check. John must verify that the bank will recognize it as a check from Tim before he hands over whatever Tim bought from him. The bank, of course, must authenticate the check before it takes the money from Tim's bank account and gives it to John.

Imagine what happens if we use shared secret keys, probably with a keyed hash (Section 8.5), to authenticate Tim's electronic checks. Tim has to share his secret key with everyone who might receive one of his checks. Sharing the key with the bank might not raise an immediate worry, since we routinely use shared secrets (PINs for ATM cards) with banks already. But if Tim shares his secret key with everyone else who might want to authenticate his electronic check (John, for example), then anyone could modify or forge one of his electronic checks. All the forger has to do is use Tim's secret key *A-80* to recompute the correct keyed hash for the forged or modified message. Section 9.1 provided examples of why banks aren't completely reliable for this, citing the occasional troubles they have with dishonest employees abusing customers' PINs.

Once we share that secret, there's nothing to prevent John or the bank from accidentally (or intentionally) misusing it, or from sharing the secret with others. Early password systems tried to limit the problem of leaked secrets by hashing the password so that its textual form wasn't easily available. This was a step in the right direction as long as attackers really needed the textual form when attacking some cryptographically protected data. But more sophisticated systems like Microsoft Windows now use the hashed password itself as the base secret for its security services (Section 12.6). Attackers can bypass these services by stealing the hashed password stored at either end of a "secured" network connection—they don't need the plaintext password.

Public key cryptography gives us back the benefits we used to get from hashed passwords: a way to do security functions with something other than the base secret itself. In public key cryptography we work with a pair of keys, the *private key* and the *public key*, which have special mathematical properties (Figure 13.1). We embed the base secret inside the private key, which is kept secret by its owner. We perform a special one-way mathematical function to construct the public key. Since the function works efficiently in one direction but not the other (like a one-way hash function), it's not practical for attackers to deduce the private key from the public key. Thus, we can safely publish and distribute the public key, even if it risks falling into the hands of attackers.

Public key algorithms use these two keys to perform *asymmetric encryption*. That is, we use one key for encryption and the other for decryption. In typical public key algorithms, we can use either key for encryption, but then we must use the other key for decryption. For example, if Tim uses his private key to encrypt something, then we must use Tim's public key to decrypt it. Likewise, if we encrypt something with Tim's public key, then nobody can decrypt it except Tim, using his private key.

Public keys always work in pairs. If we receive a message from Tim and use *his* public key to decrypt it, then we know for certain that it was encrypted with *his* private key and nobody else's. If someone else tries to encrypt something with Tim's public key, then we *cannot* decrypt it by using Tim's public key again. As with conventional secret keys, different public key pairs perform encryption differently: a message encrypted with John's private key won't be decrypted correctly by Tim's public key.

Thus, if Tim writes an electronic check and encrypts it with his private key, anyone who wants to verify its authenticity can use his public key to do so. The verification takes place entirely off-line. We don't have to perform some transaction with the bank or with some authentication server: if we have Tim's public key, then we can use **D-80**

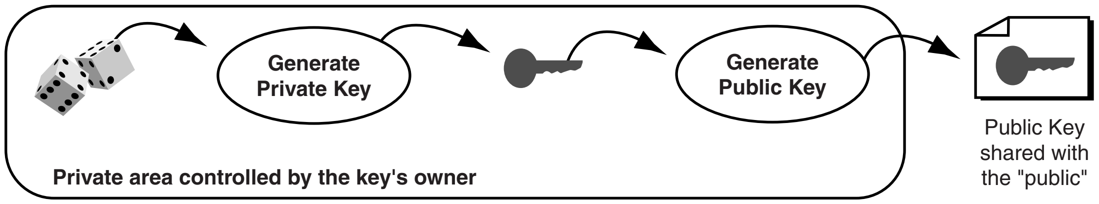

FIGURE 13.1: *Generating a public–private key pair.* First, we generate a random number and use it to choose a value for the private key. Next, we apply a special one-way function to the private key to produce the public key. Attackers can't derive the private key from the public key because the one way function is hard to reverse.

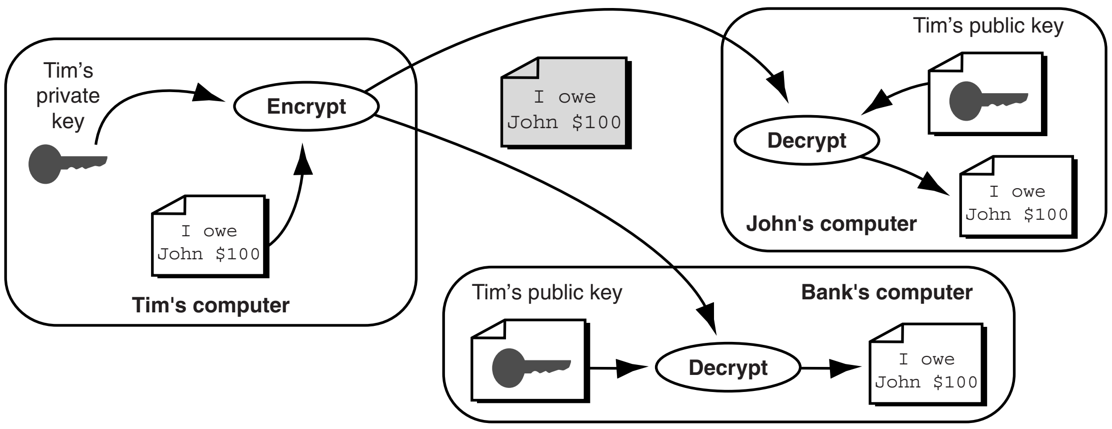

FIGURE 13.2: *"Signing" a message by encrypting it with the private key.* To authenticate his electronic check ("I owe John $100"), Tim encrypts it with his private key. The recipients (John and the bank) authenticate the message by decrypting it with a copy of Tim's public key. Attackers can't forge a message—if they encrypt it with the public key, it will yield gibberish when decrypted with the same public key.

it to authenticate his check. The encrypted electronic check serves as a primitive "digital signature" constructed by Tim to authenticate his message.

Figure 13.2 shows how we can use public key encryption to generate an electronic check that others can authenticate off-line. Tim writes the check and sends a copy to John and to his bank. Both recipients have copies of Tim's public key. Both can use the public key to verify that Tim wrote the message. Neither can modify that electronic check or create a completely different one.

Because of this, we can use public key cryptography to reduce the risks of transitive trust and of leaking base secrets. The key's owner keeps the private key secret and distributes copies of the public key to all potential recipients. Tim, John, and other bank customers generate public key pairs and share their public keys with the bank and with one another. They can validate each others' checks directly, using software residing on their own computers, and so can the bank. While this doesn't solve the problem of overdrawn bank accounts, it reduces the risk of forgery.

For authentication, we can reap the benefits of public key cryptography by replacing secret key algorithms with public key algorithms.

For example, we can replace the algorithms used in a challenge response protocol. Instead of encrypting the nonce with a shared secret key, Tim uses his private key. The authentication server can authenticate Tim using a copy of his public key. Thus, an attacker can't masquerade as Tim by simply stealing the key database, since it only contains public keys. Thus, public key authentication provides greater safety and flexibility.

13.2 THE RSA PUBLIC KEY ALGORITHM

The basic trick behind public key cryptography involves mathematical operations that are easy to do in one direction but hard to do in another. Unlike the one-way functions we use for one-way hashing, these functions have to yield numbers with a carefully tuned internal structure. This allows us to perform a series of one-way operations that won't fall apart when hit with clever mathematics.

Multiplication is a good example of how our one-way function needs to work. It's always straightforward to multiply several numbers together, but it's not always so easy to figure out what numbers were combined to produce a particular result. For example, you can easily use a pencil and paper to multiply 550 by 750 and find 412,500. On the other hand, there's no similar procedure to find which two numbers were multiplied together to yield 401,963. This is called the *factoring problem*, since it's the problem of finding what factors were multiplied together to produce a particular number. At best, one performs a search by trial and error to locate the factors. This can take a long time if the number is the product of multiplying two primes together, since primes aren't divisible by any smaller numbers, except the value *one*. Public key cryptography is based on the notion of using large enough numbers to make the trial-and-error factoring process too difficult to be practical. ***see Note 1.***

The factoring problem is at the heart of the Rivest Shamir Adelman (RSA) public key algorithm, created by Ron Rivest, Adi Shamir, and Len Adelman. RSA is arguably the most widely used public key algorithm, since every modern Web browser program uses it to protect e-commerce transactions (see Section 13.6). RSA is faster, or more flexible, or both, when compared to other public key algorithms. With RSA, we can encrypt with either the public or private key, and then decrypt with the other key. Various protocols and applications use RSA to authenticate people and computers, and to safely distribute keys used with secret key encryption.

An RSA key pair actually consists of three parts: the public part ***e***, private part ***d***, and the shared value ***N***. We construct the shared value ***N*** by choosing two very large prime numbers and multiplying them together. In practice, we often use a standard value for ***e***, like 3 or 65,537, but it can be chosen randomly as long as it meets certain mathematical restrictions. We compute ***d*** from ***e*** and the two large primes. The RSA public key consists of the shared value ***N*** along with the public part ***e***. Once we've erased the data used to produce the keys (particularly the two initial primes), only the private part ***d*** needs to be kept secret. ***see Note 2.***

RSA encryption and decryption are surprisingly simple in design. If we're encrypting with the public key, then we take the plaintext message (let's call it ***P***) and exponentiate it by the public part ***e***, and take the *modulus* relative to ***N***. In other words, we raise the number **P** to the e'th power, and then find its remainder relative to ***N***. The modulus function simply computes the integer remainder (note that the RSA shared value ***N*** is itself often called "the modulus"). Now we can look at a trivial example using small numbers. If we take ***e*** = 7, ***N*** = 527, and our secret message is 2, for example, then encryption looks like this:

$$C = P^e \bmod N \qquad \begin{aligned} C &= 2^7 \bmod 527 \\ C &= 128 \bmod 527 \\ C &= 128 \end{aligned}$$

So, the message, encrypted with the public key, is the number 128. We use essentially the same, simple computation to decrypt it, but we use the other key. The private part depends on the public part, and so we find that ***d*** = 343 if the public key's ***N*** = 527 and ***e*** = 7. The decryption takes place like this: ***see Note 3.***

$$P = C^d \bmod N \qquad \begin{aligned} P &= 128^{343} \bmod 527 \\ P &= 2 \end{aligned}$$

Now we can implement electronic checks for Tim using RSA. First, Tim must generate the three pieces of an RSA key for himself, and send a copy of his "public key" (his ***N*** and ***e*** values) to his bank.

Then he can fill out an electronic check, encode it into a large number, and then encrypt that number using his RSA private key. This takes place exactly as shown above, except that we substitute d for e in the encryption step. Now, anyone with Tim's RSA public key can decrypt his electronic check and see for themselves that Tim produced it.

13.3 ATTACKING RSA

Consider what happens if someone named Henry takes Tim's public key and tries to construct an electronic check that appears to be from Tim. Of course, Henry can encrypt an electronic check with Tim's public key, but the resulting check won't decrypt correctly when the bank uses Tim's public key to decrypt it. To forge Tim's signature, Henry can use either of two general strategies: he can figure out Tim's private key, or he can fiddle with the mathematics of the encryption to try to produce a properly encrypted check. Both techniques might work unless Tim takes the right steps to prevent them.

ATTACKING RSA KEYS

Henry can reconstruct Tim's private value d if he can find the two factors that make up the shared value N. If Tim's key is the one used in the example ($N = 527$, $e = 7$) then Henry has a reasonably easy job. He knows that the shared value N is the product of two prime numbers, and that the private key d is constructed from those two primes. So all he has to do is find which two numbers were multiplied together to produce the number 527. At worst, he could write a program to iterate through all reasonable pairs of numbers, see if their product equals 527, and stumble on the answer after trying, at most, a few dozen alternatives.

In essence, the value N is the "key" value when considering brute force attacks on RSA. For that reason, we generally refer to N as the "RSA key," and the "size" of an RSA key refers to the size of N itself. If the key size is small, like in the example, then it is vulnerable to attack. We need to determine how big our RSA key must be to resist attack. Fortunately, we can increase the size of an RSA key without changing the encryption algorithm. We only have to be sure that the software implementation will handle the key size we choose.

We can crack an RSA key by factoring it and, unfortunately, there are some very efficient techniques for factoring numbers. Unlike brute force attacks on secret keys, these factoring techniques don't have to consider every possible value of the key individually. In fact, classical factorization techniques require $N^{1/2}$ steps, which means that the average attack space contains *half* as many bits as the RSA key. And the story gets even worse. ***A-81*** ***see Note 4.***

When RSA was introduced in 1977, Ron Rivest predicted that it would take 40 quadrillion years to factor an RSA key containing 125 decimal digits (416 bits), and whimsically offered $100 to the first person to do it. But his prediction proved inaccurate by approximately 40 quadrillion years when a team of researchers broke an RSA message encrypted with a 129-digit key in 1994. To be fair, Rivest's estimate might have been plausible in 1977. Back in 1977, factoring relied on classical techniques, so a 416-bit RSA key presented a 208-bit average attack space. A brute force attack of that size was impractical back then and remains impractical today. ***see Note 5.***

Although Rivest's challenge started as a bit of a joke, it became serious as mathematicians made incredible progress on the factoring problem. A few years later, RSA Data Security, the company that held the patent rights to the RSA algorithm (now RSA Security), officially established the RSA Factoring Challenge. The challenge consisted of a series of messages encrypted with RSA keys of different lengths, and each challenge was named according to the number of decimal digits in the key. Thus, the RSA-129 challenge contained a message encrypted with a 129-digit (429-bit) RSA key. The successful attack on RSA-129, led by Arjen Lenstra of Bellcore, used a factoring technique called the quadratic sieve. ***see Note 6.***

Between 1991 and 1999, teams of mathematicians and computer scientists cracked seven RSA challenge messages by factoring larger and larger numbers. RSA-129 was actually the last time that the quadratic sieve was used to crack an RSA challenge. Later successes relied on a different technique: the number field sieve. The quadratic sieve worked efficiently against smaller RSA keys. When key sizes (and challenges) exceed the 429-bit keys of the RSA-129 challenge, the number field sieve becomes the practical choice.

In 1999, a new team (though including a few people from the RSA-129 effort) used the number field sieve to crack RSA-155, a message encrypted with a 512-bit RSA key. The attack was comparable to a 63-bit average attack space. The 512-bit key was a particularly important milestone, since that used to be a standard key size used for e-commerce on the World Wide Web. RSA Security used the occasion to recommend that RSA keys be at least 768 bits long. However, a survey of e-commerce servers the following year found that over 25% of them were still using keys of 512 bits or less. ***see Note 7.***

RSA's press release is right: the principal countermeasure against factoring attacks is to use larger RSA key sizes. Typically we use heuristics to estimate the number of computational steps a sieve will require to factor an RSA key of a given size; we can use the same heuristic to estimate the average attack space of an RSA key. Table 13.1 compares the average attack space for common RSA key sizes against other cryptographic authentication techniques. Typical secret key sizes rarely approach 200 bits, and the corresponding average attack space is usually only one or two bits less than that. Since the average attack space for an RSA key is dramatically smaller than the key size, we must obviously choose a correspondingly larger key size. Modern RSA keys are often more than a thousand bits long, and in some cases may contain over 2,000 bits. ***D-81*** ***see Note 8.***

TABLE 13.1: *Comparing Public Keys with Other Average Attack Spaces*

Typical Use	Average Attack Space
Eight-character personally chosen password	22.7 bits
56-bit DES (ANSI X9.9, other DES tokens, encryption, etc.)	54 bits
SecurID one-time password token	63 bits
512-bit public keys for digital signatures	63 bits
768-bit key—RSA Security's minimum RSA key size, 1999	76 bits
1024-bit public keys	86 bits
2048-bit high-security public keys	116 bits
128-bit Advanced Encryption Standard	127 bits

Factoring attacks look incredibly powerful in comparison to the straightforward brute force attacks posed by secret key algorithms, and that by itself might make some people uneasy. If attacks have made so much progress against public keys so quickly, then what's to prevent this "trend" from continuing until public key cryptography is worthless? We can answer that question in several ways.

First, we look at why the factoring attack seems to be the best way to attack an RSA private key. Mathematicians have found a few arguments to support this belief. Attempts to crack the key by attacking other elements of the encryption, like the private part d of the private key, turn out to require as much work as attacks directly on the N value itself. And, while mathematicians haven't been able to prove conclusively that all attacks on the RSA private key will require factoring of N, they have proven it for something similar, the *Rabin cryptosystem*. However, the Rabin system isn't as practical to use, so instead we rely on their similarities to give us confidence in the safety of RSA encryption. ***see Note 9.***

Second, we can consider why it's likely that factoring will remain a hard enough problem to protect our data. For centuries, mathematicians have acknowledged that factoring is a hard computational problem. No mathematician has ever posed a credible argument that there might be a shortcut for factoring very large numbers. There has never been a "last theorem" in which a noted mathematician posed a solution to the factoring problem but omitted the details. There was tremendous progress in factoring in the late 20th century, thanks to the development of the quadratic sieve and number field sieve. However, there is no reason to expect that further developments will make the number field sieve look as obsolete as the classical techniques look today.

Finally, we must keep the risks against RSA in perspective. None of the arguments here absolve RSA of mathematical risk, but that's the nature of modern cryptography. Keep in mind what we know (and don't know) about the strength of secret key encryption. Although we have heaps of practical experience with secret key algorithms, we have no way to prove mathematically that our algorithms are strong. The same is true of RSA.

ATTACKING DIGITAL SIGNATURES

When discussing secret key cryptography we identified some ways to attack the encrypted data without knowing the key. Similar attacks exist against public key algorithms, including RSA. These attacks rely on the mathematical structure of RSA—they work by deriving mathematical factors that will affect encrypted RSA data in controlled, malicious ways.

For example, Henry is still trying to forge one of Tim's checks, and he has adapted one of these mathematical attacks to do so. The crucial trick is that Henry must get Tim to sign a message that looks like gibberish ("I'm just testing something in the system, Tim. Hey, look at the message yourself. Look as closely as you want. It doesn't mean anything!"). If Henry can get Tim to sign such a message, he can exploit the mathematical properties of modular exponentiation to construct a forged message signed by Tim.

This is called a *chosen message attack*, and public key mathematics provides several ways to do it. In one approach, Henry first constructs a message containing random bits, and encrypts it with Tim's public key. Then Henry combines the encrypted result with a forged check of Tim's. This yields the gibberish that Henry asks Tim to sign. Remember that when Tim signs the message, he is encrypting it with his private key. After Tim signs the message, Henry combines it with an inverted form of his original random bits, which mathematically deducts those bits from the signature. This leaves Henry with a forged check signed by Tim.

A-82

see Note 10.

Fortunately, there is a single, well-known technique that prevents the chosen message attack from working: compute a hash of the message being signed and encrypt the hash only. Figure 13.3 shows

D-82

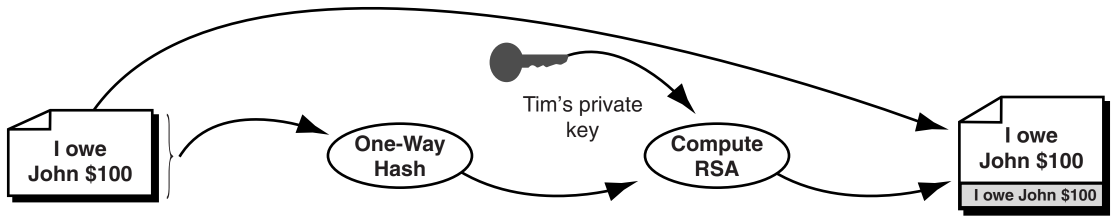

FIGURE 13.3: *Applying an RSA digital signature to a message.* To sign his message, Tim first computes the one-way hash of the text of his message. Then he encrypts the resulting hash value using his private key. Finally, he attaches this digital signature value to the original message. Anyone can verify the contents of the message by checking this digital signature value.

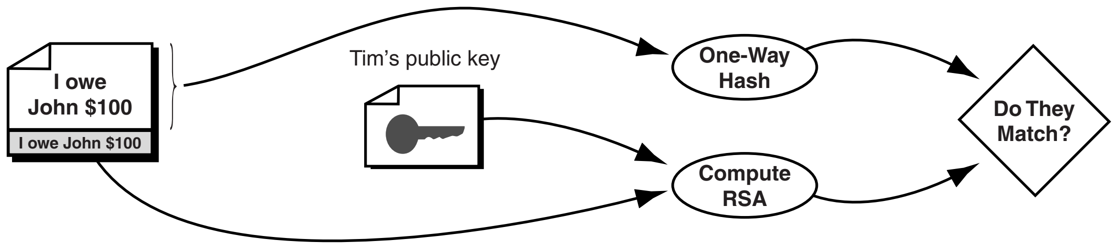

FIGURE 13.4: *Checking the RSA digital signature on a message.* To verify that Tim actually signed the message we received, we must compare the hash of the message against the hash Tim encrypted with his private key. We decrypt that hash using Tim's public key and compare the result with the hash we get from the message's text.

how to construct a digital signature using a hash function. To sign his check, Tim first computes the one-way hash value of its text. Next, Tim encrypts the hash value using RSA with his private key.

If John, the bank, or anyone else, wants to authenticate the check that Tim signed, they use Tim's public key as shown in Figure 13.4. John takes the text and computes the one-way hash, just at Tim did. John does *not* include the digital signature itself in the hashed text, since Tim didn't include that value, either. (Of course, Tim couldn't include the digital signature value in the hash, since he couldn't possibly know its value until after he'd computed the hash —a circular reference he can't possibly resolve.) Then John decrypts the digital signature value by applying RSA with Tim's *public* key. If the check is authentic, then the decrypted value will match the hash value.

Chosen message attacks rely on classic mathematical symmetries, like the associative, commutative, and distributive properties we learned in early arithmetic. Henry's attack against an unhashed signature succeeds because he can do parts of the computation piecemeal, and combine those parts in clever ways. If we add a hashing step to the signature process, we destroy part of the symmetry of the mathematical operations and thus prevent the attack from working.

13.4 THE DIGITAL SIGNATURE STANDARD

By 1990, RSA digital signatures had found their way into a few successful applications, most notably Lotus Notes. However, agencies of the U.S. government could not use digital signatures based on public keys because there were no accepted federal standards for such signatures. To fill this gap, NIST proceeded to develop such a standard. However, the standard NIST proposed in 1991 was *not* based on RSA despite its popularity with commercial users. The actual reason for this wasn't clear. One suggested rationale was that RSA was protected by a U.S. patent, and NIST wanted a standard unencumbered by licensing and fees. ***see Note 11.***

But regardless of the reason, the FIPS Digital Signature Standard (DSS) is not based on RSA or factorization. Of course, factoring isn't the only mathematical problem appropriate for public key cryptography. A different problem, the *discrete logarithm problem*, provides the public key algorithm used in the DSS. When we exponentiate a number, we raise it to a particular power, which in turn involves a series of multiplications. As we noted before, multiplication is a straightforward operation, so exponentiation is fairly straightforward, too. However, it is a bit more difficult to invert the exponentiation, that is, to take its logarithm.

There are a number of practical mathematical techniques for computing logarithms (in fact, tables of logarithms were one of the principal 19th-century goals of mechanical computation). However, there are ways of modifying the problem so that the classic techniques don't work. In particular, we can exponentiate a number and then compute its modulus relative to some base value. The modulus operation discards the high order bits of the exponentiation, and this makes it hard to undo the exponentiation. To make things even more difficult, we compute the exponentiation and the modulus against large prime numbers.

The mathematician and cryptographer Taher ElGamal published a paper in 1985 describing how to apply the exponentiation problem to encryption and in particular to digital signatures. ElGamal had been a student of Martin Hellman, the cryptographer who co-developed the first successful public key algorithm: Diffie-Hellman. Like ElGamal's work, Diffie-Hellman is based on the discrete logarithm problem. Unlike the developers of RSA, however, ElGamal had published a paper describing his technique long before he considered filing a patent application, so his technique was not eligible for a patent.

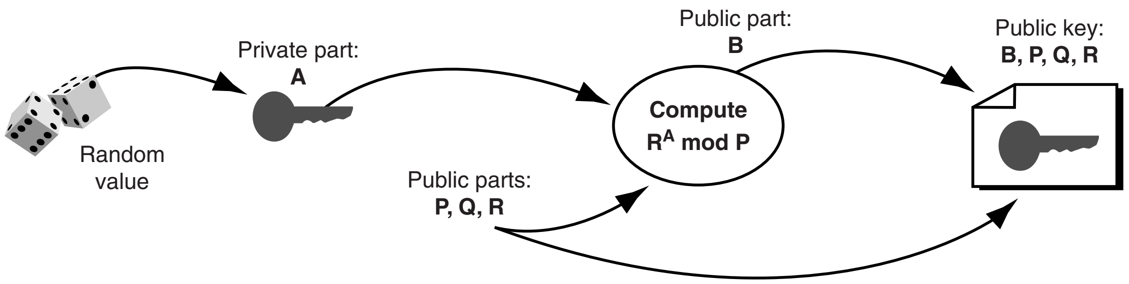

FIGURE 13.5: *Creating a public key pair for exponential techniques like DSS.* Here is how we produce a public key for exponential public key methods DSS, ElGamal, or Diffie-Hellman. First, we select a large (512-bit) prime **P**. Next we select two numbers **Q**, a smaller prime, and **R**, both based on the value of **P**. We pick **A** at random and keep it secret. We compute the value of **B** as shown here. The values of **B**, **P**, **Q**, and **R** make up the public key. **Q** is used in digital signatures.

Figure 13.5 shows how we construct a public key pair for DSS and similar, logarithm-based techniques. The computation involves the exponentiation and modulus pattern we saw in the RSA algorithm, but here we use it to construct the key pair. First, we choose three public numbers: a large prime, **P**, and two other numbers, **Q** and **R**, related to P. For our private secret, we choose the random secret value **A**. We produce the public key by raising **R** to the **A**-th power, and taking its modulus relative to **P**. Since the modulus operation factors out the upper digits of the exponentiation, there's no straightforward way to take the result (called **B** here) and find the value of **A** that produced it.

From the outside, the procedure looks reasonably similar to the RSA procedure of signing a message: it requires a hash of the message's text, the private part of the key pair, and some of the public part of the key pair. Inside, however, the procedure is far more complicated. Figure 13.6 sketches the procedure. First, we hash the function as usual, but then we must generate a random value, **K**, that we use to produce a "check value" used in the verification procedure. The signature consists of two pieces of information: the signature value computed from the hash, and the check value. The verification procedure applies a series of modular exponentiations and modular inverses to verify the signature. Unfortunately, the process is very slow to execute as well as being complicated. *see Note 12.*

An interesting feature of the mechanism is that it can only be used for digital signatures. With RSA, the computation can encrypt

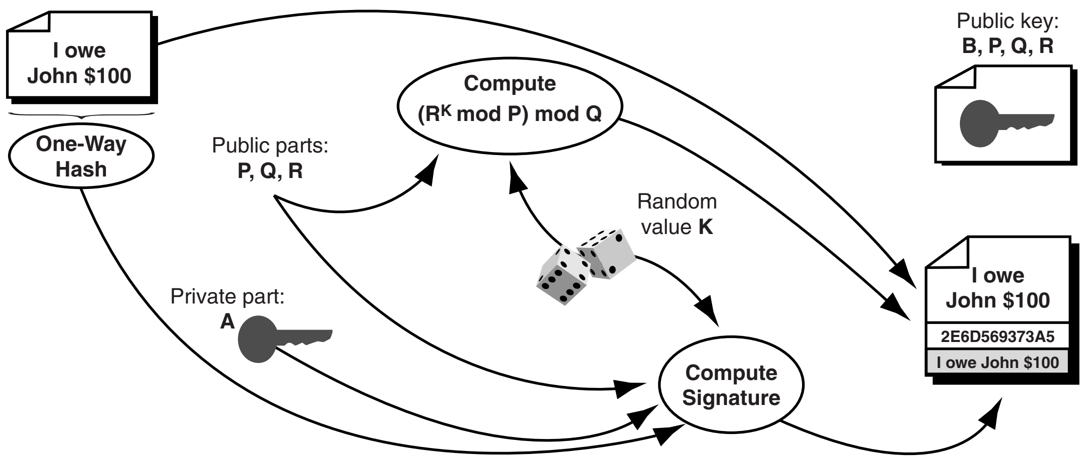

FIGURE 13.6: *Applying a DSS digital signature to a message.* As with RSA, Tim first hashes his message. Then he chooses a random value **K** and computes a check value to be used to validate the signature. Then he uses the key's private part **A** in conjunction with **K** and the public key components to compute the signature value. Then Tim attaches the check value and signature value to the message and delivers it. The recipient authenticates the message by computing a series of equations that use the public key, the signature, and the check value.

data as well as sign documents. This made ElGamal's scheme more appealing in 1991, since the U.S. government heavily restricted exports of cryptographic systems at that time. The DSS, a variation of the ElGamal scheme, was adopted as a federal standard in December 1994.

The basic approach for attacking a DSS key, or any other public key that relies on modular exponentiation, is to apply a relatively efficient algorithm for finding the discrete logarithm. One such algorithm is called the index calculus method, and it is very similar to the number field sieve technique used to factor RSA keys. In fact, the performance estimates are so close that we use the same average attack space estimates for both RSA and DSS keys. *see Note 13.*

13.5 CHALLENGE RESPONSE REVISITED

The previous sections have used electronic checks as an authentication problem, but it's a bit different from the problem of verifying who is talking to a particular computer at a particular instant. The digitally signed document poses some eternal statement, like "Tim pays John $100 on Flag Day, 2001," or something like that. It

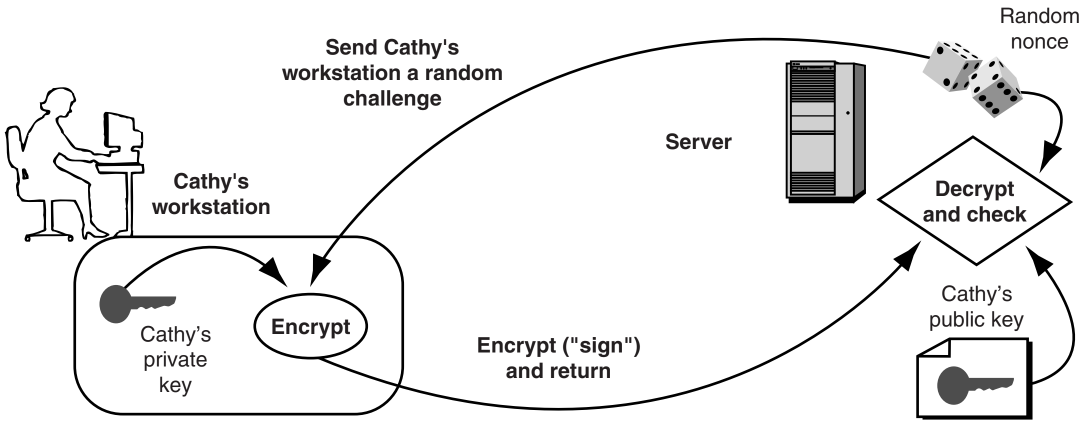

FIGURE 13.7: *Trivial challenge response using public key cryptography.* The server sends Cathy a nonce in response to her request to log on. To authenticate herself, Cathy encrypts the nonce with her private key, which essentially generates its digital signature. The server compares her response to the original nonce's value, after decrypting it with her public key. The server can authenticate Cathy without having to contact another server for authoritative information.

doesn't say that Tim is present at any particular computer at any particular moment. We can use public key cryptography to do that as well.

Figure 13.7 illustrates a simple protocol to authenticate someone using public key cryptography and a challenge response strategy. When Cathy, for example, tries to log in to the server, it transmits a random challenge. To prove her identity, Cathy encrypts the random challenge and sends it back. The server verifies her identity using her public key: her decrypted response should match the challenge that it sent.

This approach provides true off-line authentication, since the server only needs a copy of Cathy's public key. The server does not need to contact a separate authentication server to verify Cathy's identity.

As shown earlier with shared secret authentication, we can use variants of this approach to produce one-time passwords without a challenge (Section 9.3). For example, Cathy could encrypt a clock value, and submit that along with her user name when logging on. The server would decrypt the clock value and verify that it's within some acceptable window. However, Cathy would have to keep her computer's clock very closely synchronized with the server's clock; otherwise the server would have to tolerate a fairly wide clock skew. This in turn could make the system vulnerable to a sniff and replay attack (Attack A-79 in Section 12.5).

As with one-time passwords, we could also use a counter instead of a clock. However, this would not allow off-line authentication. Servers would need to share information about the counter's value, and that would create the need for an on-line authentication protocol.

Although we used challenge response with some success in Chapter 10, this particular approach poses an interesting problem: it might make Cathy vulnerable to signature forgeries. or other attacks. Since public key mathematics are vulnerable to attacks of the type described in the previous section (Attack A-81), we want to avoid using our private key to encrypt arbitrary data received from others.

LOCKOUT FORTEZZA AUTHENTICATION PROTOCOL

A practical solution to these problems was developed to use with the Fortezza card, a special cryptographic module developed by the U.S. government. The Fortezza card was a standard PCMCIA card developed by the NSA to provide a general-purpose cryptographic capability for government and commercial applications. The Fortezza card contained implementations of the DSS algorithm introduced in Section 13.4 and the SHA introduced in Section 8.5.

The authentication protocol was originally called the "Tessera Authentication Protocol," since "Tessera" was the original name for the Fortezza card. The protocol was named "LOCKOut Fortezza" when it was released as a product. The protocol used SHA and DSS, since those were the algorithms provided on the Fortezza card. The basic design could have accommodated RSA as well. The protocol was developed in 1994 for government agencies that required a strong and reasonably automatic authentication mechanism based on public key cryptography, notably the defense finance community. End users installed PC card readers on their workstations, and sites installed the protocol on their firewalls. *see Note 14.*

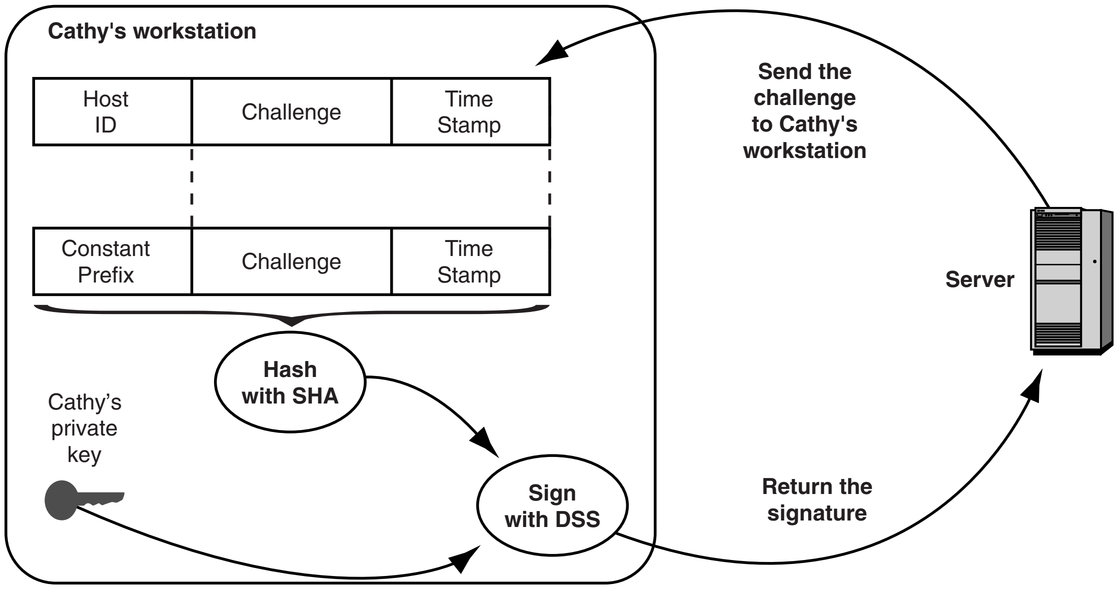

FIGURE 13.8: *The LOCKOut Fortezza authentication protocol.* This protocol uses challenge response with DSS to provide authentication using a public key. Cathy sends her user name to the server, which replies with a host ID, a random challenge, and a time stamp. Cathy constructs her response and sends its signature back to the server. The server uses Cathy's public key to verify that the signature matches the expected hash result.

As with Microsoft's challenge response, the process is embedded entirely in software, since it exchanges too much data for people to type in. Figure 13.8 illustrates the protocol. Cathy begins the authentication process by providing her user name to the server. The server responds with a server identifier known to Cathy, a 12-bit time stamp, and an 80-bit challenge. The server identifier is a numerical value chosen by Cathy's workstation software when she enrolled with the server and provided the server with a copy of her public key. After checking the time stamp for freshness, Cathy's workstation constructs a response. The workstation combines an 80-bit constant prefix, the server's challenge, and the server's time stamp, and then computes their DSS digital signature. Her response consists of the digital signature.

Note that the figure has simplified the DSS picture in comparison to Figure 13.6. Subsequent figures do not show the different DSS public components separately, or separate the check value from the

DSS signature value. Instead, the DSS "public key" includes all public data by implication, and the DSS "signature" includes both the signature itself and the separate check value it requires.

To verify Cathy's response, the server retrieves a copy of the challenge and time stamp it sent her, constructs a copy of the response, including the constant prefix, and computes the hash on it. Then the server extracts Cathy's public key and checks the signature value response against the computed hash. The two should match.

There is actually a situation in which the protocol allows for this match to fail: if Cathy has logged on "under duress," she can secretly signal that fact to the server. One of the problems with off-line authentication is that it can be hard to revoke credentials; we will examine this further in Section 14.6. So, consider a dramatic *A-83* scenario in which Cathy is taken hostage by an international spy or gangster and forced to log on to a critical military system with her Fortezza card. She can give her workstation a secret signal that makes a subtle, hard-to-detect change in the digital signature. *D-83* Specifically, it changes the constant prefix from one value to another. If the server's first attempt to verify the signature fails, the server can try again with the duress version of the prefix. If that one succeeds, the server can take appropriate actions. For example, the server can send a silent alarm to security officers, and it can make changes to Cathy's access permissions that take into account the fact that she may be in danger and not in a position to act responsibly. As noted in Section 1.4, it can be relatively easy to provide duress signals, but it is difficult to use them effectively.

In Kerberos, the logon process generally authenticated the client workstations to servers, and vice versa as well. The LOCKOut Fortezza protocol also provided this feature through a simple extension to the server's challenge. To authenticate itself, the server combines the challenge data with the normal prefix value and signs the result. Then it sends this signature along with the rest of the challenge message. Upon receipt, Cathy can authenticate the server by checking the signature against the server's public key.

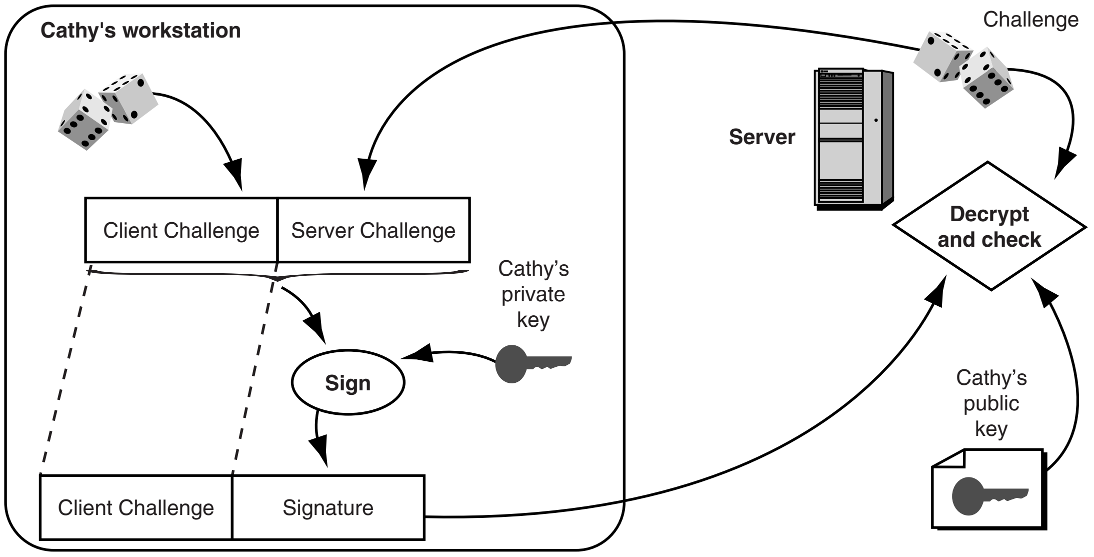

FIGURE 13.9: *FIPS 196 one-way authentication.* The server sends Cathy a challenge in response to her request to log on. Cathy generates a challenge herself, combines it with the server's challenge, and generates a digital signature with her private key. Then she sends the digital signature plus her own challenge back as her response. The server combines its challenge and her challenge, and uses them to verify Cathy's signature with her public key.

FIPS 196 AUTHENTICATION

In 1997, NIST published a federal standard for public key authentication: FIPS 196. Like the LOCKOut Fortezza protocol, it would authenticate the client to the server, and could optionally authenticate the server to the client as well, using public key cryptography. However, FIPS 196 did not define a protocol for interoperability; instead it defined the general transactions required and left details up to the implementers. *see Note 15.*

Figure 13.9 illustrates the FIPS 196 protocol. The process starts with a request by the client workstation that yields a random challenge from the server. To respond, Cathy generates her own challenge and combines it with the server's challenge. She computes her digital signature using the two challenges, and her response contains her challenge along with the digital signature. Unlike LOCKOut Fortezza, FIPS 196 relies entirely on nonces and does not use time stamps.

To verify Cathy's response, the server retrieves the challenge it sent to her along with a copy of her public key. The server reconstructs the message that Cathy signed, her client challenge combined with the server's challenge, and verifies the signature against that message.

Although these authentication protocols were used in the defense community, they never achieved widespread acceptance. Clearly, its association with the Fortezza card was a detriment to the LOCKOut Fortezza protocol, since Fortezza proved too expensive for widespread use. Also, Fortezza had a bad reputation in the computer security community because it was associated with escrowed encryption. But the most important reason was probably the emergence of the Secure Sockets Layer protocol, which provided the same features and more and deployed them successfully on the World Wide Web.

13.6 Secure Sockets Layer

In 1991, a British computer scientist named Tim Berners-Lee developed the World Wide Web at INRIA in Switzerland. The original purpose was to simplify the exchange of technical papers, particularly in physics, stored on Internet servers around the world. Mark Andressen and a team of software developers at the University of Illinois went on in 1993 to produce Mosaic, a general-purpose graphical "browser" for visiting Web pages. Mosaic integrated the World Wide Web protocols with the ability to display graphic images and to support other data retrieval protocols, including the File Transfer Protocol, and Gopher, a menu-driven protocol.

A year later, Andressen left the university to found Netscape, a company seeking to provide software so that businesses could use the Web. Security was naturally an essential part of this, and the young Netscape company arranged to use the public key algorithms from RSA. *see Note 16.*

Moreover, Netscape's designers looked closely at the capabilities of public key technology, and of "hybrid" strategies that combined it with secret key encryption. The result was a very clever technical approach that put the bulk of public key management problems on the server. This approach was incorporated into the *Secure Sockets Layer* (SSL) protocol. *see Note 17.*

SSL is undoubtedly the most common public key application on Earth. In 1995, Jim Barksdale, president of Netscape Communications, estimated that there had been approximately 20 million copies of Netscape's SSL-enabled browser distributed worldwide. Here are features of SSL:

- **Safety and convenience:** SSL establishes a strong, cryptographically protected connection between a client workstation and a server, and usually requires no participation by the workstation's operator.
- **Server authenticity:** SSL authenticates the server to ensure that the client is talking to the correct computer. This authentication uses "public key certificates," described in Chapter 14.
- **Automatic client authentication:** A workstation operator can authenticate to an SSL-enabled server using the operator's own public key pair, if the server is set up to accept it.
- **Extensibility:** SSL can use most encryption algorithms and hashing algorithms. The client and server can automatically choose the best algorithms from those they both support.

Although SSL is extensible, there's a relatively standard implementation that's used almost universally for consumer sites on the World Wide Web. The typical site uses RSA public key encryption to establish a set of shared secret keys used with RC4 for encryption and MD5 for integrity protection. A typical server will authenticate itself to clients, but does not support SSL's built-in client authentication. However, some servers do support the built-in client authentication. The next two sections describe these alternatives.

Figure 13.10 illustrates the essential concept behind the SSL protocol. Cathy contacts the server, and the server responds by sending a copy of its public key. Cathy then generates a base secret, encrypts it with the server's public key, and sends it back to the server. If we think of this in Kerberos terms, the public key allows Cathy to act as her own KDC, generating her own ticket to send to the server. As with Kerberos tickets, the shared secret lets Cathy's workstation exchange encrypted and integrity-protected messages with the server. The only difference is that Cathy hasn't really been authenticated in this process.

The complete protocol is, of course, more complicated than this. Part of the complication comes from flexibility: by breaking the protocol up into separate layers and messages, the designers made it easier to support a whole range of options for public key algorithms and other security services. Other complications arose from attacks: as people in the community studied SSL, they saw troubling possibilities. Between 1994 and 1997, Netscape developed three versions of SSL, with each newer version seeking to fix the problems found in the previous one. The last Netscape version, 3.0, is the one described here. Arguably, the safety of SSL 3.0 is comparable to other peer-reviewed protocols like Kerberos and RADIUS. *see Note 18.*

SSL is designed to provide cryptographic protection to messages traveling on a TCP connection, that is, a reliable, bidirectional connection established using standard Internet protocols. SSL provides another layer atop TCP that protects messages against forgery, modification, and sniffing. When a pair of hosts first establish an SSL connection, the messages they exchange are still embedded in SSL transport messages, but no encryption takes place. The hosts must negotiate the use of cryptographic services via SSL's handshake protocol.

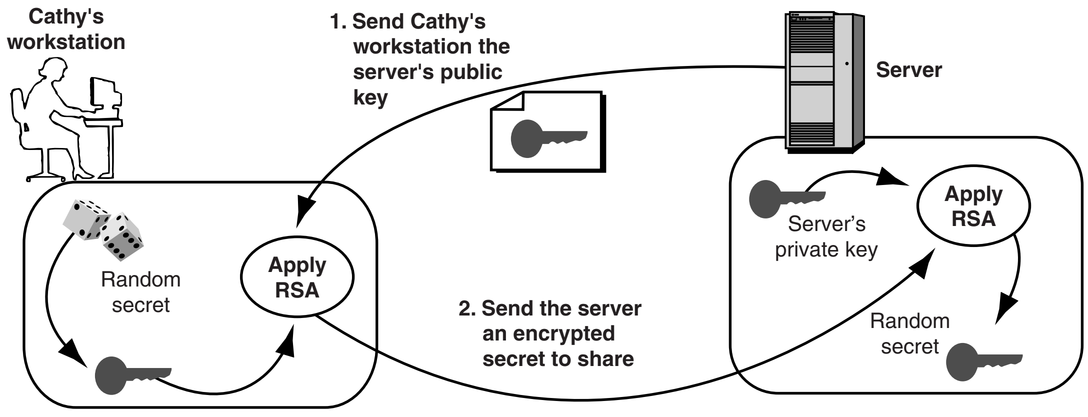
Cathy's workstation
1. Send Cathy's workstation the server's public key
Server
Random secret
Apply RSA
Server's private key
Apply RSA
2. Send the server an encrypted secret to share
Random secret

FIGURE 13.10: *Secure Sockets Layer lets the client and server share a secret key.* When Cathy contacts the server and asks for a secure connection, the server sends her a copy of its public key. Cathy randomly generates a secret to share with the server, and encrypts that secret with the server's public key. Only the server's private key will be able to decrypt that secret. Cathy sends the encrypted secret to the server. The server decrypts the secret, and they use the shared secret to encrypt and authenticate subsequent messages, just like with Kerberos session keys. Note that this protocol does not authenticate Cathy to the server.

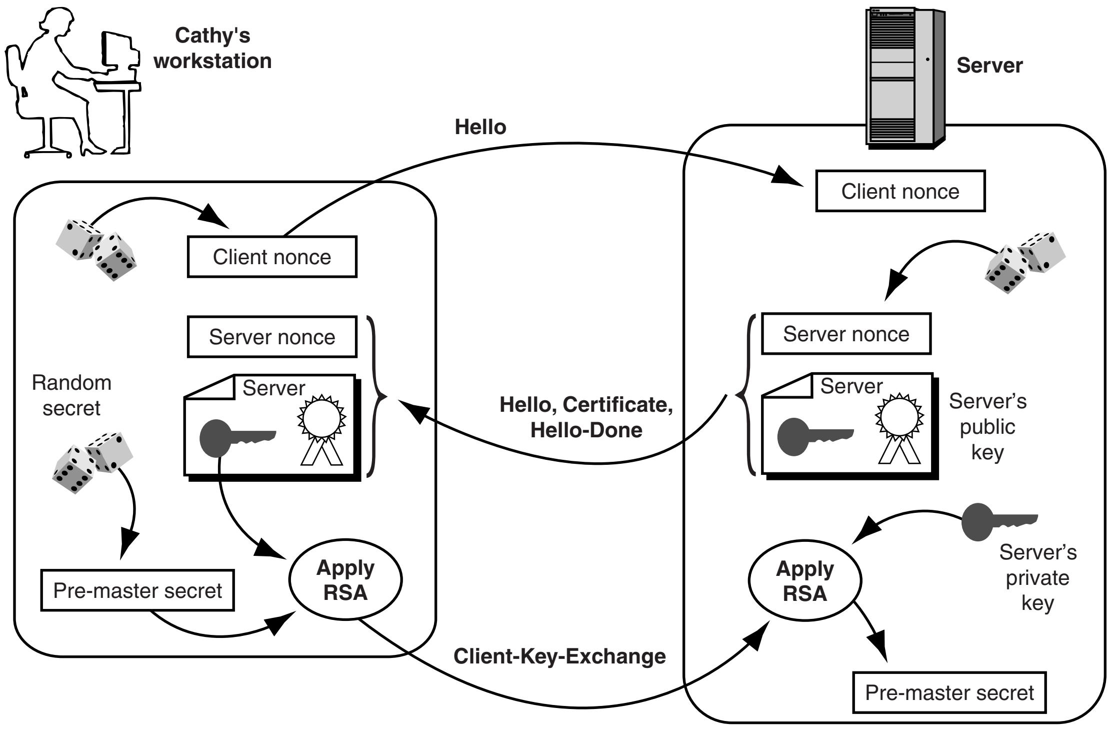

FIGURE 13.11: *Details of the SSL handshake protocol, Part 1.* Cathy's workstation starts the process with a Hello message containing random data. The server responds with a three-message sequence that provides another nonce and the server's public key (embedded in a "certificate"). The client responds with a three-message sequence, but only one shown here. The Client-Key-Exchange message transmits a shared secret, encrypted with the server's public key. The server decrypts the secret with its private key, yielding the "pre-master secret" used to construct shared keys to cryptographically protect the traffic between the client and server. The client does not need any preestablished crypto keys to establish the shared secret with the server.

ESTABLISHING KEYS WITH SSL

Figure 13.11 illustrates SSL's handshake protocol. The server and client take turns sending a series of messages to the other. Once the exchanges are finished, the cryptographic protections and shared keys are in place. Here is the process:

- Client says: "Hello."

  Once the client has opened the connection, it generates 28 bytes of random data and transmits them to the server in a "Hello" message. This is the first message transmitted in Figure 13.11.

- Server says: “Hello, Certificate, Hello-Done.”

  Upon receipt of the client’s “Hello,” the server responds with a sequence of three messages, illustrated by the second message transmitted in Figure 13.11. The first is a “Hello” to echo the one from the client, containing 28 random bytes that the server generated itself.

  The next message is a “Certificate” message, which contains the server’s public key embedded in a “public key certificate” data structure. The client needs to check the authenticity of the server’s certificate before using its public key. Certificates and certificate validation are described in Chapter 14.

  The final message, the “Hello-Done,” simply tells the client that the server is through with this step of the handshake process.

- Client: “Client-Key-Exchange, Change-Cipher-Spec, Finished.”

  The client responds with a sequence of three messages. The “Client-Key-Exchange” message is the heart of SSL: it transmits the secret from the client to the server, encrypted with the server’s public key. In SSL, the shared secret is called a *pre-master secret*. As shown in Figure 13.11, the client extracts the public key from the server’s certificate and performs RSA encryption. Upon receipt, the server retrieves the pre-master secret by

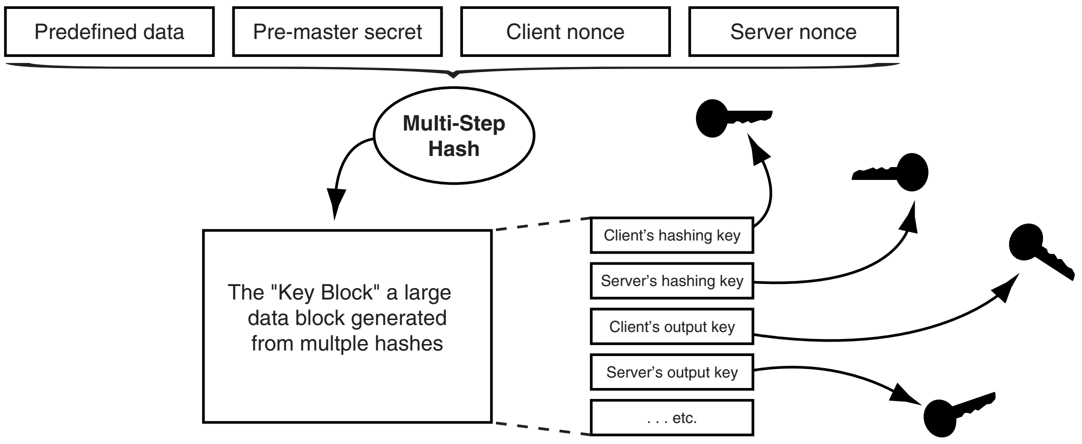

FIGURE 13.12: *Constructing SSL crypto keys.* SSL constructs all of its working keys by hashing the nonces with the pre-master secret and other, predefined data to construct a large data block called the Key Block. To create the individual keys, we subdivide the Key Block into separate sections, one per key. The server and client construct a matching set of keys from the Key Block.

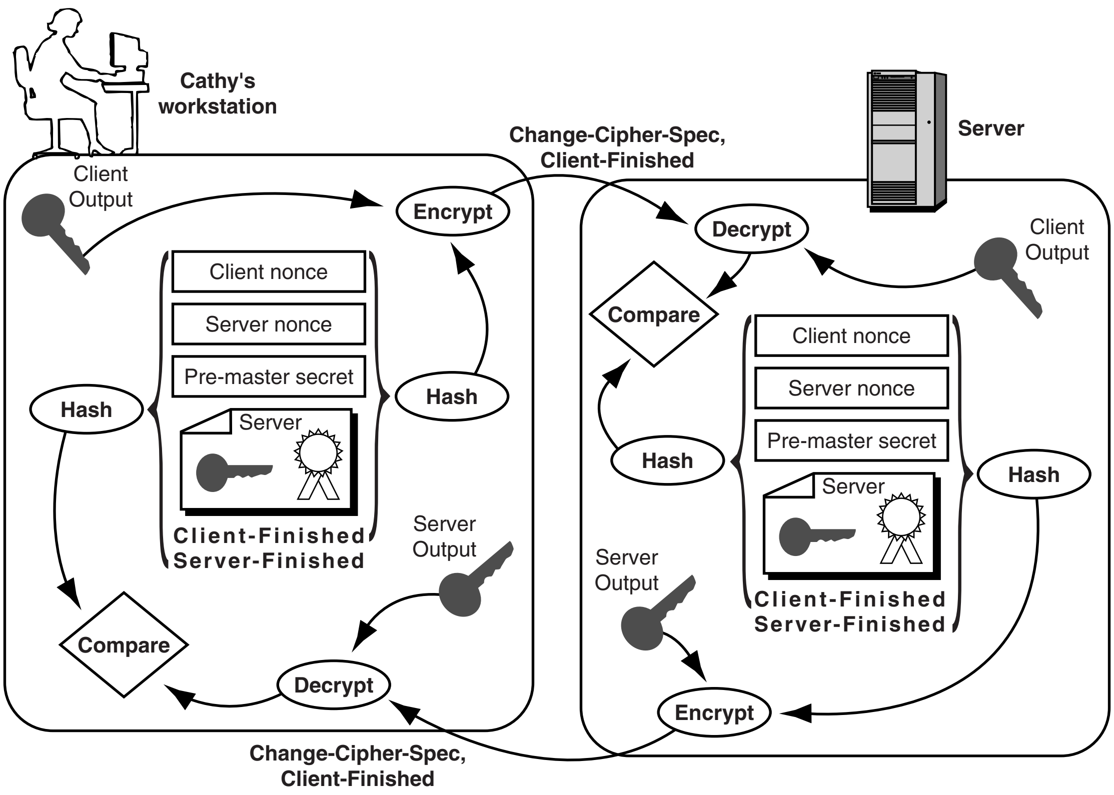

FIGURE 13.13: *Details of the SSL handshake protocol, Part 2.* At this point, the client and server are ready to switch to the newly negotiated keys, and to verify that each has successfully retrieved the same keys as the other. The Change-Cipher-Spec messages each signal when the host is ready to switch to the new keys. The Client-Finished and Server-Finished messages transmit the hash of all the messages received by the respective hosts, so the other host can verify that both received the same series of messages.

applying RSA with its private key. Once this step is finished, both the client and server share the secret and non-secret information they need to generate the SSL session keys. Figure 13.12 illustrates the key generation process.

Figure 13.13 shows the processing of the remaining two messages in this sequence. The client's "Change-Cipher-Spec" message simply signals the point at which the client will start using the keys established during this handshake. The "Client-Finished" message transmits hashes of all of the data shared during the handshake process. The hash is computed twice: once with

MD5 and once with SHA. The resulting hashes are encrypted and integrity protected using the keys just established.

Upon receipt, the server uses the client's output key to decrypt the "Finished" message. Then the server checks the message contents against its own hashes of the handshake messages. If all went correctly, both should match. This tells the server that the client has set up its keys correctly.

- Server: "Change-Cipher-Spec, Finished."

  This final sequence echoes the last two message types sent by the client, but from the server's perspective. The "Change-Cipher-Spec" message indicates that the server will now start using the keys that were just established. All subsequent data will be encrypted and integrity protected with the new keys.

  The "Server-Finished" message transmits almost the same data hash as the client's earlier message, except that now we must include the hash of the "Client-Finished" message. The server constructs the appropriate hashes, encrypts the message with the "server output" keys, and sends the message to the client.

Upon receipt of the "Server-Finished" message, the client decrypts it with the "server output" keys, and compares its contents with the client's own computation of the correct hashes. If they match, then the client is confident that the server possesses the corresponding private key, and thus must be authentic.

AUTHENTICATION WITH TYPICAL SSL

Now that the client has established the authenticity of the server, how will the server establish the authenticity of the client? In practice, servers tend to use typical authentication methods, and they often take advantage of the fact that passwords can't be sniffed on an encrypted SSL connection. Here is a summary of techniques that can authenticate a client over an SSL connection:

- User names and passwords, self-selected
- E-mail addresses and passwords
- Information about credit card accounts
- Externally assigned user names and passwords

- Quizzes with cultural secrets belonging to a select group
- One-time password systems

An unfortunate legacy of U.S. export restrictions is that many desktops run older versions of SSL that use only 40 bits of secrecy in their encryption keys. While most browsers distributed today provide at least 128 bits of secrecy, the older browsers are still used. Often, the users aren't even aware of the fact that they are using relatively weak encryption to protect their private information.

SSL CLIENT AUTHENTICATION

Clients can also authenticate themselves to servers by using SSL's built-in client authentication. This protocol uses a public key pair belonging to the workstation's user, and performs a challenge response transaction to verify that the user really possesses the appropriate private key.

The process works as shown in Figure 13.14. It is integrated into the SSL handshake protocol, in the server's final message and the client's response to it. The server asks for the client's public key in the form of a certificate, and the client replies with both the certificate and a response encrypted with the corresponding private key.

The server prompts for client-side authentication by including a "Certificate-Request" message along with its "Hello" message sequence. This message includes information about the certificates that the server is willing to accept. If the client can't accommodate the server's expectations, it sends back a "No-Certificate" warning. Upon receipt, the server can either reject the connection or require some other form of authentication.

In response, the client includes two additional messages in its message sequence. The first message is a "Certificate" message containing the client's certificate. The other message is a "Certificate-Verify" message that serves as the "response" portion of a challenge response protocol. For a "challenge," the message uses the computed hashes of the previous messages in the handshake protocol. This is similar to the "Finished" messages shown in Figure 13.13. However, the "Certificate-Verify" computes a digital signature from the hashes, and transmits those results to the server.

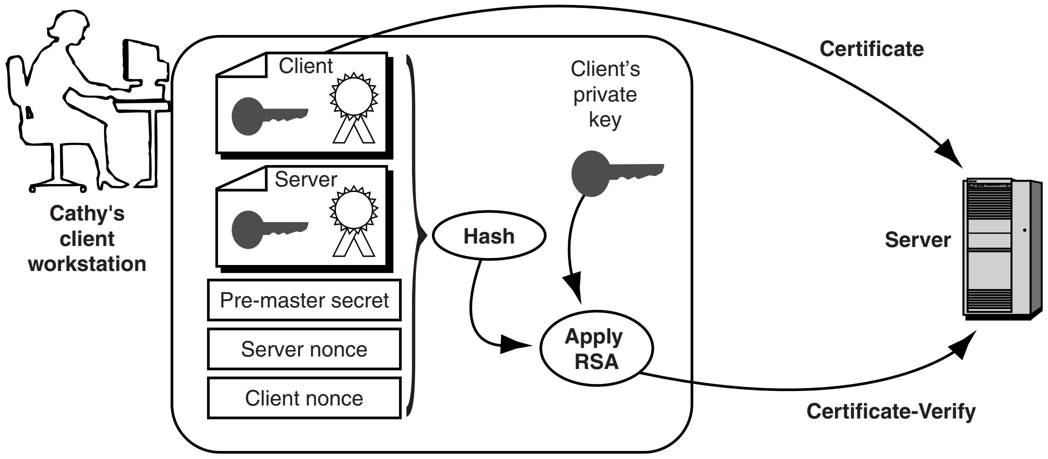

FIGURE 13.14: *SSL client authentication.* Cathy's client sends two separate messages in order to authenticate her. The Certificate message transmits her personal public key certificate. Later, she sends a Certificate-Verify message, containing a hash covering the messages she exchanged with the server during the SSL handshake, encrypted with her private key.

From a security standpoint, the SSL client authentication protocol is reasonably strong. The "challenge" incorporates nonces recently generated by both the server and client, so replay isn't a practical attack. More reasonable attacks would go after either the certificates or after the client's private key. These attacks, and their defenses, are described in Chapters 14 and 15 respectively.

13.7 PUBLIC KEYS AND BIOMETRICS

It may seem to some readers of Section 7.7 on biometric encryption that public keys could "solve" the biometrics problem. As with so many technologies, public key technology simply rearranges the problem without really solving it. Here is a look at two possible applications of public keys to biometrics and their limitations.

- **Protect a biometric reading from sniffing**

  We could create a public key for a biometric system and embed that public key in all biometric readers. Whenever a biometric signature is collected and sent in for authentication, we can encrypt it first with the public key. This will prevent attackers from sniffing the biometric.

While this approach would reduce the risk of someone intercepting a biometric reading collected by the system, it does not prevent an attacker from collecting a victim's biometric in any number of other ways. While there are valid privacy concerns that justify the encryption of biometrics, we cannot assume that such protections will keep biometric readings a secret. This simply gives people a false sense of security, letting them assume that their biometrics can serve as safe base secrets.

- **Authenticate the source of a biometric reading**

  We can take things a step further and embed private keys in biometric readers. The reader's key would be used to digitally sign the biometric so that the system has confidence that the reading originated from an acceptable reader. This gives the system a way to detect spoofed biometrics, if we assume that the readers themselves can reliably detect spoofing.

  Of course, this approach substitutes the problems of biometrics with the problems of managing the private keys that must be installed in all of the biometric readers. If the proprietor is willing to spend the administrative resources necessary to manage those private keys, it may make more sense to simply assign the keys directly to individuals instead of assigning them to biometric readers. This is an architectural trade-off that individual sites and system managers must make.

While we can indeed use SSL's approach to public key cryptography to protect biometric readings from interception, we cannot use it to prevent spoofing. For that, we need to associate a base secret with the transmitter of the biometric reading, so we are sure the reading originates from a trustworthy source. Otherwise, an attacker with a copy of the public key and a convincing version of the victim's biometric reading can masquerade as that victim. As noted in Chapter 7, there are lots of ways to capture biometric readings even without the victim's cooperation.

13.8 SUMMARY TABLES

TABLE 13.2: *Attack Summary*

Attack	Security Problem	Prevalence	Attack Description
A-80. Shared key misuse for forgery	Masquerade as someone else	Trivial	Secret key that was shared with a trusted party is used to forge a message
A-81. Factoring an RSA key	Recover hidden information (RSA key)	Sophisticated	Factor the RSA composite and deduce the private key
A-82. Chosen message attack	Recover hidden information (RSA key)	Common	Construct a message with a special mathematical structure. Victim signs it, and result can be transformed into one for a different message
A-83. Duress logon with private key	Masquerade as someone else	Physical and Sophisticated	Attacker forces the owner of a private key to log on, then uses the established session

TABLE 13.3: *Defense Summary*

Defense	Foils Attacks	Description
D-80. Public key encryption	A-80. Shared key misuse for forgery	Use public key encryption that anyone can verify and associate with the owner of a specific private key
D-81. Significantly larger RSA key sizes	A-81. Factoring an RSA key	Generate RSA keys containing thousands of bits; certainly more than 1,000 bits
D-82. Hashed digital signature	A-82. Chosen message attack	Construct digital signatures by encrypting the result of a one-way hash
D-83. Duress signature	A-83. Duress logon with private key	Construct a special form digital signature indicating that user is under duress