

8.9 WEB SECURITY

We have just studied two important areas where security is needed: communications and e-mail. You can think of these as the soup and appetizer. Now it is time for the main course: Web security. The Web is where most of the Trudies hang out nowadays and do their dirty work. In the following sections we will look at some of the problems and issues relating to Web security.

Web security can be roughly divided into three parts. First, how are objects and resources named securely? Second, how can secure, authenticated connections be established? Third, what happens when a Web site sends a client a piece of executable code? After looking at some threats, we will examine all these issues.

8.9.1 Threats

One reads about Web site security problems in the newspaper almost weekly. The situation is really pretty grim. Let us look at a few examples of what has already happened. First, the home page of numerous organizations has been attacked and replaced by a new home page of the crackers' choosing. (The popular press calls people who break into computers "hackers," but many programmers reserve that term for great programmers. We prefer to call these people "crackers.") Sites that have been cracked include Yahoo, the U.S. Army, the CIA, NASA, and the New York Times. In most cases, the crackers just put up some funny text and the sites were repaired within a few hours.

Now let us look at some much more serious cases. Numerous sites have been brought down by denial-of-service attacks, in which the cracker floods the site with traffic, rendering it unable to respond to legitimate queries. Often the attack is mounted from a large number of machines that the cracker has already broken into (DDoS attacks). These attacks are so common that they do not even make the news any more, but they can cost the attacked site thousands of dollars in lost business.

In 1999, a Swedish cracker broke into Microsoft's Hotmail Web site and created a mirror site that allowed anyone to type in the name of a Hotmail user and then read all of the person's current and archived e-mail.

In another case, a 19-year-old Russian cracker named Maxim broke into an e-commerce Web site and stole 300,000 credit card numbers. Then he approached the site owners and told them that if they did not pay him \$100,000, he would post all the credit card numbers to the Internet. They did not give in to his blackmail, and he indeed posted the credit card numbers, inflicting great damage to many innocent victims.

In a different vein, a 23-year-old California student e-mailed a press release to a news agency falsely stating that the Emulex Corporation was going to post a large quarterly loss and that the C.E.O. was resigning immediately. Within hours,

the company's stock dropped by 60%, causing stockholders to lose over \$2 billion. The perpetrator made a quarter of a million dollars by selling the stock short just before sending the announcement. While this event was not a Web site break-in, it is clear that putting such an announcement on the home page of any big corporation would have a similar effect.

We could (unfortunately) go on like this for many pages. But it is now time to examine some of the technical issues related to Web security. For more information about security problems of all kinds, see (Anderson, 2001; Garfinkel with Spafford, 2002; and Schneier, 2000). Searching the Internet will also turn up vast numbers of specific cases.

8.9.2 Secure Naming

Let us start with something very basic: Alice wants to visit Bob's Web site. She types Bob's URL into her browser and a few seconds later, a Web page appears. But is it Bob's? Maybe yes and maybe no. Trudy might be up to her old tricks again. For example, she might be intercepting all of Alice's outgoing packets and examining them. When she captures an HTTP *GET* request headed to Bob's Web site, she could go to Bob's Web site herself to get the page, modify it as she wishes, and return the fake page to Alice. Alice would be none the wiser. Worse yet, Trudy could slash the prices at Bob's e-store to make his goods look very attractive, thereby tricking Alice into sending her credit card number to "Bob" to buy some merchandise.

One disadvantage to this classic man-in-the-middle attack is that Trudy has to be in a position to intercept Alice's outgoing traffic and forge her incoming traffic. In practice, she has to tap either Alice's phone line or Bob's, since tapping the fiber backbone is fairly difficult. While active wiretapping is certainly possible, it is a certain amount of work, and while Trudy is clever, she is also lazy. Besides, there are easier ways to trick Alice.

DNS Spoofing

For example, suppose Trudy is able to crack the DNS system, maybe just the DNS cache at Alice's ISP, and replace Bob's IP address (say, 36.1.2.3) with her (Trudy's) IP address (say, 42.9.9.9). That leads to the following attack. The way it is supposed to work is illustrated in Fig. 8-46(a). Here (1) Alice asks DNS for Bob's IP address, (2) gets it, (3) asks Bob for his home page, and (4) gets that, too. After Trudy has modified Bob's DNS record to contain her own IP address instead of Bob's, we get the situation of Fig. 8-46(b). Here, when Alice looks up Bob's IP address, she gets Trudy's, so all her traffic intended for Bob goes to Trudy. Trudy can now mount a man-in-the-middle attack without having to go to the trouble of tapping any phone lines. Instead, she has to break into a DNS server and change one record, a much easier proposition.

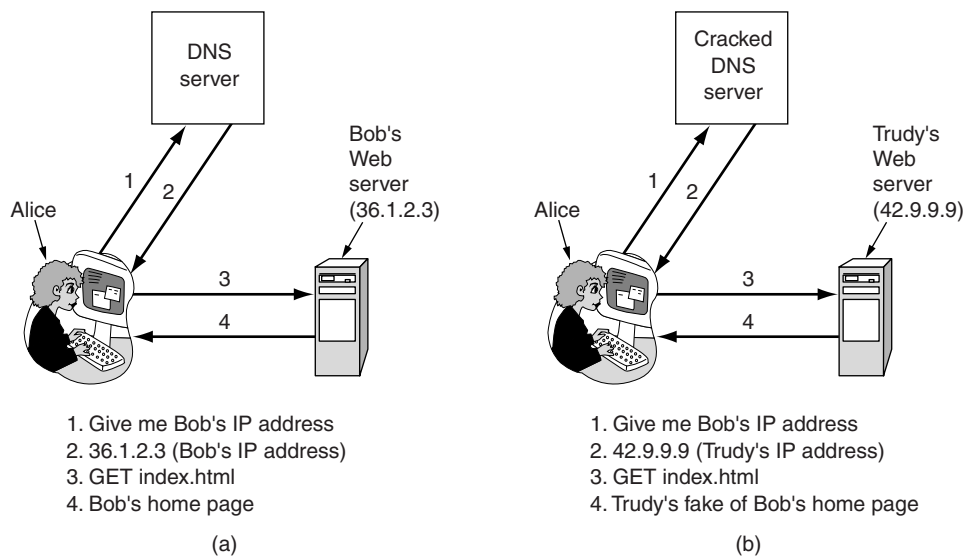


Figure 8-46. (a) Normal situation. (b) An attack based on breaking into DNS and modifying Bob's record.

How might Trudy fool DNS? It turns out to be relatively easy. Briefly summarized, Trudy can trick the DNS server at Alice's ISP into sending out a query to look up Bob's address. Unfortunately, since DNS uses UDP, the DNS server has no real way of checking who supplied the answer. Trudy can exploit this property by forging the expected reply and thus injecting a false IP address into the DNS server's cache. For simplicity, we will assume that Alice's ISP does not initially have an entry for Bob's Web site, *bob.com*. If it does, Trudy can wait until it times out and try later (or use other tricks).

Trudy starts the attack by sending a lookup request to Alice's ISP asking for the IP address of *bob.com*. Since there is no entry for this DNS name, the cache server queries the top-level server for the *com* domain to get one. However, Trudy beats the *com* server to the punch and sends back a false reply saying: "*bob.com* is 42.9.9.9," where that IP address is hers. If her false reply gets back to Alice's ISP first, that one will be cached and the real reply will be rejected as an unsolicited reply to a query no longer outstanding. Tricking a DNS server into installing a false IP address is called **DNS spoofing**. A cache that holds an intentionally false IP address like this is called a **poisoned cache**.

Actually, things are not quite that simple. First, Alice's ISP checks to see that the reply bears the correct IP source address of the top-level server. But since Trudy can put anything she wants in that IP field, she can defeat that test easily since the IP addresses of the top-level servers have to be public.

Second, to allow DNS servers to tell which reply goes with which request, all requests carry a sequence number. To spoof Alice's ISP, Trudy has to know its current sequence number. The easiest way to learn the current sequence number is for Trudy to register a domain herself, say, *trudy-the-intruder.com*. Let us assume its IP address is also 42.9.9.9. She also creates a DNS server for her newly-hatched domain, *dns.trudy-the-intruder.com*. It, too, uses Trudy's 42.9.9.9 IP address, since Trudy has only one computer. Now she has to make Alice's ISP aware of her DNS server. That is easy to do. All she has to do is ask Alice's ISP for *foobar.trudy-the-intruder.com*, which will cause Alice's ISP to find out who serves Trudy's new domain by asking the top-level *com* server.

With *dns.trudy-the-intruder.com* safely in the cache at Alice's ISP, the real attack can start. Trudy now queries Alice's ISP for *www.trudy-the-intruder.com*. The ISP naturally sends Trudy's DNS server a query asking for it. This query bears the sequence number that Trudy is looking for. Quick like a bunny, Trudy asks Alice's ISP to look up Bob. She immediately answers her own question by sending the ISP a forged reply, allegedly from the top-level *com* server saying: "*bob.com* is 42.9.9.9". This forged reply carries a sequence number one higher than the one she just received. While she is at it, she can also send a second forgery with a sequence number two higher, and maybe a dozen more with increasing sequence numbers. One of them is bound to match. The rest will just be thrown out. When Alice's forged reply arrives, it is cached; when the real reply comes in later, it is rejected since no query is then outstanding.

Now when Alice looks up *bob.com*, she is told to use 42.9.9.9, Trudy's address. Trudy has mounted a successful man-in-the-middle attack from the comfort of her own living room. The various steps to this attack are illustrated in Fig. 8-47. To make matters worse, this is not the only way to spoof DNS. There are many other ways as well.

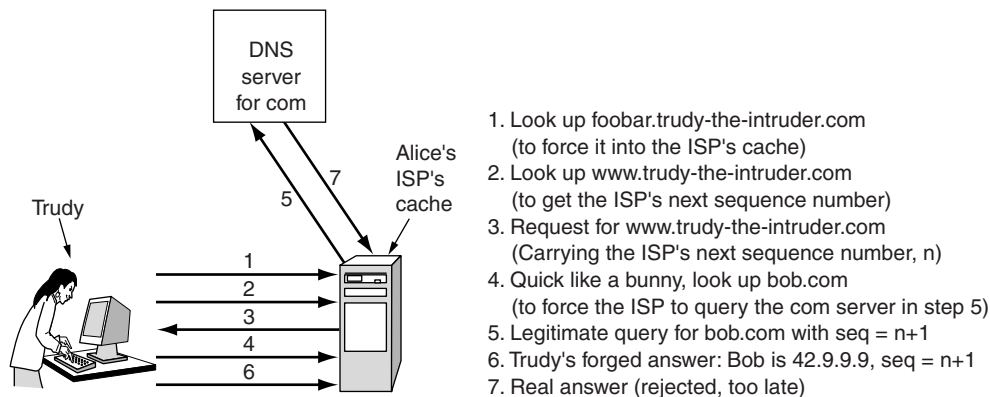


Figure 8-47. How Trudy spoofs Alice's ISP.

Secure DNS

This one specific attack can be foiled by having DNS servers use random IDs in their queries rather than just counting, but it seems that every time one hole is plugged, another one turns up. The real problem is that DNS was designed at a time when the Internet was a research facility for a few hundred universities and neither Alice, nor Bob, nor Trudy was invited to the party. Security was not an issue then; making the Internet work at all was the issue. The environment has changed radically over the years, so in 1994 IETF set up a working group to make DNS fundamentally secure. This project is known as **DNSsec (DNS security)**; its output is presented in RFC 2535. Unfortunately, DNSsec has not been fully deployed yet, so numerous DNS servers are still vulnerable to spoofing attacks.

DNSsec is conceptually extremely simple. It is based on public-key cryptography. Every DNS zone (in the sense of Fig. 7-4) has a public/private key pair. All information sent by a DNS server is signed with the originating zone's private key, so the receiver can verify its authenticity.

DNSsec offers three fundamental services:

1. Proof of where the data originated.
2. Public key distribution.
3. Transaction and request authentication.

The main service is the first one, which verifies that the data being returned has been approved by the zone's owner. The second one is useful for storing and retrieving public keys securely. The third one is needed to guard against playback and spoofing attacks. Note that secrecy is not an offered service since all the information in DNS is considered public. Since phasing in DNSsec is expected to take several years, the ability for security-aware servers to interwork with security-ignorant servers is essential, which implies that the protocol cannot be changed. Let us now look at some of the details.

DNS records are grouped into sets called **RRSets (Resource Record Sets)**, with all the records having the same name, class and type being lumped together in a set. An RRSet may contain multiple *A* records, for example, if a DNS name resolves to a primary IP address and a secondary IP address. The RRSets are extended with several new record types (discussed below). Each RRSet is cryptographically hashed (e.g., using MD5 or SHA-1). The hash is signed by the zone's private key (e.g., using RSA). The unit of transmission to clients is the signed RRSet. Upon receipt of a signed RRSet, the client can verify whether it was signed by the private key of the originating zone. If the signature agrees, the data are accepted. Since each RRSet contains its own signature, RRSets can be cached anywhere, even at untrustworthy servers, without endangering the security.

DNSsec introduces several new record types. The first of these is the *KEY* record. This records holds the public key of a zone, user, host, or other principal,

the cryptographic algorithm used for signing, the protocol used for transmission, and a few other bits. The public key is stored naked. X.509 certificates are not used due to their bulk. The algorithm field holds a 1 for MD5/RSA signatures (the preferred choice), and other values for other combinations. The protocol field can indicate the use of IPsec or other security protocols, if any.

The second new record type is the *SIG* record. It holds the signed hash according to the algorithm specified in the *KEY* record. The signature applies to all the records in the RRSet, including any *KEY* records present, but excluding itself. It also holds the times when the signature begins its period of validity and when it expires, as well as the signer's name and a few other items.

The DNSsec design is such that a zone's private key can be kept off-line. Once or twice a day, the contents of a zone's database can be manually transported (e.g., on CD-ROM) to a disconnected machine on which the private key is located. All the RRSets can be signed there and the *SIG* records thus produced can be conveyed back to the zone's primary server on CD-ROM. In this way, the private key can be stored on a CD-ROM locked in a safe except when it is inserted into the disconnected machine for signing the day's new RRSets. After signing is completed, all copies of the key are erased from memory and the disk and the CD-ROM are returned to the safe. This procedure reduces electronic security to physical security, something people understand how to deal with.

This method of presigning RRSets greatly speeds up the process of answering queries since no cryptography has to be done on the fly. The trade-off is that a large amount of disk space is needed to store all the keys and signatures in the DNS databases. Some records will increase tenfold in size due to the signature.

When a client process gets a signed RRSet, it must apply the originating zone's public key to decrypt the hash, compute the hash itself, and compare the two values. If they agree, the data are considered valid. However, this procedure begs the question of how the client gets the zone's public key. One way is to acquire it from a trusted server, using a secure connection (e.g., using IPsec).

However, in practice, it is expected that clients will be preconfigured with the public keys of all the top-level domains. If Alice now wants to visit Bob's Web site, she can ask DNS for the RRSet of *bob.com*, which will contain his IP address and a *KEY* record containing Bob's public key. This RRSet will be signed by the top-level *com* domain, so Alice can easily verify its validity. An example of what this RRSet might contain is shown in Fig. 8-48.

Now armed with a verified copy of Bob's public key, Alice can ask Bob's DNS server (run by Bob) for the IP address of *www.bob.com*. This RRSet will be signed by Bob's private key, so Alice can verify the signature on the RRSet Bob returns. If Trudy somehow manages to inject a false RRSet into any of the caches, Alice can easily detect its lack of authenticity because the *SIG* record contained in it will be incorrect.

However, DNSsec also provides a cryptographic mechanism to bind a response to a specific query, to prevent the kind of spoof Trudy managed to pull off

Domain name	Time to live	Class	Type	Value
bob.com.	86400	IN	A	36.1.2.3
bob.com.	86400	IN	KEY	3682793A7B73F731029CE2737D...
bob.com.	86400	IN	SIG	86947503A8B848F5272E53930C...

Figure 8-48. An example RRSet for *bob.com*. The *KEY* record is Bob's public key. The *SIG* record is the top-level *com* server's signed hash of the *A* and *KEY* records to verify their authenticity.

in Fig. 8-47. This (optional) antispoofing measure adds to the response a hash of the query message signed with the respondent's private key. Since Trudy does not know the private key of the top-level *com* server, she cannot forge a response to a query Alice's ISP sent there. She can certainly get her response back first, but it will be rejected due to its invalid signature over the hashed query.

DNSsec also supports a few other record types. For example, the *CERT* record can be used for storing (e.g., X.509) certificates. This record has been provided because some people want to turn DNS into a PKI. Whether this actually happens remains to be seen. We will stop our discussion of DNSsec here. For more details, please consult RFC 2535.

Self-Certifying Names

Secure DNS is not the only possibility for securing names. A completely different approach is used in the **Secure File System** (Mazières et al., 1999). In this project, the authors designed a secure, scalable, worldwide file system, without modifying (standard) DNS and without using certificates or assuming the existence of a PKI. In this section we will show how their ideas could be applied to the Web. Accordingly, in the description below, we will use Web terminology rather than the file system terminology used in the paper. But to avoid any possible confusion, while this scheme *could* be applied to the Web to achieve high security, it is not currently in use and would require substantial software changes to introduce it.

We start out by assuming that each Web server has a public/private key pair. The essence of the idea is that each URL contains a cryptographic hash of the server's name and public key as part of the URL. For example, in Fig. 8-49 we see the URL for Bob's photo. It starts out with the usual *http* scheme followed by the DNS name of the server (*www.bob.com*). Then comes a colon and 32-character hash. At the end is the name of the file, again as usual. Except for the hash, this is a standard URL. With the hash, it is a **self-certifying URL**.

The obvious question is: What is the hash for? The hash is computed by concatenating the DNS name of the server with the server's public key and running

Server SHA-1 (Server, Server's Public key) File name
 http://www.bob.com:2g5hd8bfjkc7mf6hg8dgany23xds4pe6/photos/bob.jpg

Figure 8-49. A self-certifying URL containing a hash of server's name and public key.

the result through the SHA-1 function to get a 160-bit hash. In this scheme, the hash is represented as a sequence of 32 digits and lower-case letters, with the exception of the letters “1” and “o” and the digits “1” and “0”, to avoid confusion. This leaves 32 digits and letters over. With 32 characters available, each one can encode a 5-bit string. A string of 32 characters can hold the 160-bit SHA-1 hash. Actually, it is not necessary to use a hash; the key itself could be used. The advantage of the hash is to reduce the length of the name.

In the simplest (but least convenient) way to see Bob's photo, Alice just types the string of Fig. 8-49 to her browser. The browser sends a message to Bob's Web site asking him for his public key. When Bob's public key arrives, the browser concatenates the server name and public key and runs the hash algorithm. If the result agrees with the 32-character hash in the secure URL, the browser is sure it has Bob's public key. After all, due to the properties of SHA-1, even if Trudy intercepts the request and forges the reply, she has no way to find a public key that gives the expected hash. Any interference from her will thus be detected. Bob's public key can be cached for future use.

Now Alice has to verify that Bob has the corresponding private key. She constructs a message containing a proposed AES session key, a nonce, and a timestamp. She then encrypts the message with Bob's public key and sends it to him. Since only Bob has the corresponding private key, only Bob is able to decrypt the message and send back the nonce encrypted with the AES key. Upon receiving the correct AES-encrypted nonce, Alice knows she is talking to Bob. Also, Alice and Bob now have an AES session key for subsequent *GET* requests and replies.

Once Alice has Bob's photo (or any Web page), she can bookmark it, so she does not have to type in the full URL again. Furthermore, the URLs embedded in Web pages can also be self certifying, so they can be used by just clicking on them, but with the additional security of knowing that the page returned is the correct one. Other ways to avoid the initial typing of the self-certifying URLs are to get them over a secure connection to a trusted server or have them present in X.509 certificates signed by CAs.

Another way to get self-certifying URLs would be to connect to a trusted search engine by typing in its self-certifying URL (the first time) and going through the same protocol as described above, leading to a secure, authenticated connection to the trusted search engine. The search engine could then be queried, with the results appearing on a signed page full of self-certifying URLs that could be clicked on without having to type in long strings.

Let us now see how well this approach stands up to Trudy's DNS spoofing. If Trudy manages to poison the cache of Alice's ISP, Alice's request may be falsely delivered to Trudy rather than to Bob. But the protocol now requires the recipient of an initial message (i.e., Trudy) to return a public key that produces the correct hash. If Trudy returns her own public key, Alice will detect it immediately because the SHA-1 hash will not match the self-certifying URL. If Trudy returns Bob's public key, Alice will not detect the attack, but Alice will encrypt her next message, using Bob's key. Trudy will get the message, but she will have no way to decrypt it to extract the AES key and nonce. Either way, all spoofing DNS can do is provide a denial-of-service attack.

8.9.3 SSL—The Secure Sockets Layer

Secure naming is a good start, but there is much more to Web security. The next step is secure connections. We will now look at how secure connections can be achieved.

When the Web burst into public view, it was initially used for just distributing static pages. However, before long, some companies got the idea of using it for financial transactions, such as purchasing merchandise by credit card, on-line banking, and electronic stock trading. These applications created a demand for secure connections. In 1995, Netscape Communications Corp, the then-dominant browser vendor, responded by introducing a security package called **SSL (Secure Sockets Layer)** to meet this demand. This software and its protocol is now widely used, also by Internet Explorer, so it is worth examining in some detail.

SSL builds a secure connection between two sockets, including

1. Parameter negotiation between client and server.
2. Mutual authentication of client and server.
3. Secret communication.
4. Data integrity protection.

We have seen these items before, so there is no need to elaborate on them.

The positioning of SSL in the usual protocol stack is illustrated in Fig. 8-50. Effectively, it is a new layer interposed between the application layer and the transport layer, accepting requests from the browser and sending them down to TCP for transmission to the server. Once the secure connection has been established, SSL's main job is handling compression and encryption. When HTTP is used over SSL, it is called **HTTPS (Secure HTTP)**, even though it is the standard HTTP protocol. Sometimes it is available at a new port (443) instead of the standard port (80), though. As an aside, SSL is not restricted to being used only with Web browsers, but that is its most common application.

The SSL protocol has gone through several versions. Below we will discuss only version 3, which is the most widely used version. SSL supports a variety of different algorithms and options. These options include the presence or absence

Application (HTTP)
Security (SSL)
Transport (TCP)
Network (IP)
Data link (PPP)
Physical (modem, ADSL, cable TV)

Figure 8-50. Layers (and protocols) for a home user browsing with SSL.

of compression, the cryptographic algorithms to be used, and some matters relating to export restrictions on cryptography. The last is mainly intended to make sure that serious cryptography is used only when both ends of the connection are in the United States. In other cases, keys are limited to 40 bits, which cryptographers regard as something of a joke. Netscape was forced to put in this restriction in order to get an export license from the U.S. Government.

SSL consists of two subprotocols, one for establishing a secure connection and one for using it. Let us start out by seeing how secure connections are established. The connection establishment subprotocol is shown in Fig. 8-51. It starts out with message 1 when Alice sends a request to Bob to establish a connection. The request specifies the SSL version Alice has and her preferences with respect to compression and cryptographic algorithms. It also contains a nonce, R_A , to be used later.

Now it is Bob's turn. In message 2, Bob makes a choice among the various algorithms that Alice can support and sends his own nonce, R_B . Then in message 3, he sends a certificate containing his public key. If this certificate is not signed by some well-known authority, he also sends a chain of certificates that can be followed back to one. All browsers, including Alice's, come preloaded with about 100 public keys, so if Bob can establish a chain anchored at one of these, Alice will be able to verify Bob's public key. At this point Bob may send some other messages (such as a request for Alice's public-key certificate). When Bob is done, he sends message 4 to tell Alice it is her turn.

Alice responds by choosing a random 384-bit **premaster key** and sending it to Bob encrypted with his public key (message 5). The actual session key used for encrypting data is derived from the premaster key combined with both nonces in a complex way. After message 5 has been received, both Alice and Bob are able to compute the session key. For this reason, Alice tells Bob to switch to the new cipher (message 6) and also that she is finished with the establishment subprotocol (message 7). Bob then acknowledges her (messages 8 and 9).

However, although Alice knows who Bob is, Bob does not know who Alice is (unless Alice has a public key and a corresponding certificate for it, an unlikely situation for an individual). Therefore, Bob's first message may well be a request for Alice to log in using a previously established login name and password. The

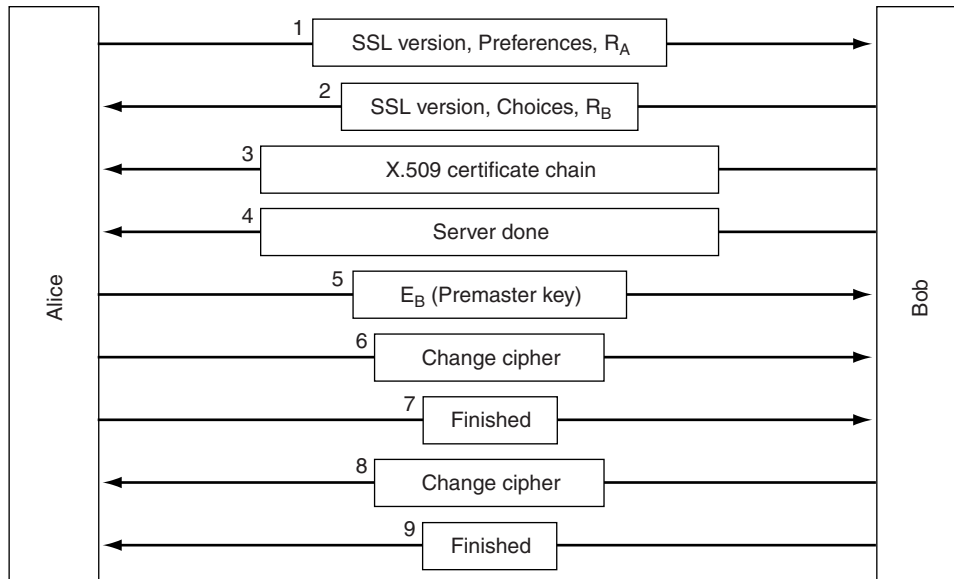


Figure 8-51. A simplified version of the SSL connection establishment subprotocol.

login protocol, however, is outside the scope of SSL. Once it has been accomplished, by whatever means, data transport can begin.

As mentioned above, SSL supports multiple cryptographic algorithms. The strongest one uses triple DES with three separate keys for encryption and SHA-1 for message integrity. This combination is relatively slow, so it is mostly used for banking and other applications in which the highest security is required. For ordinary e-commerce applications, RC4 is used with a 128-bit key for encryption and MD5 is used for message authentication. RC4 takes the 128-bit key as a seed and expands it to a much larger number for internal use. Then it uses this internal number to generate a keystream. The keystream is XORed with the plaintext to provide a classical stream cipher, as we saw in Fig. 8-14. The export versions also use RC4 with 128-bit keys, but 88 of the bits are made public to make the cipher easy to break.

For actual transport, a second subprotocol is used, as shown in Fig. 8-52. Messages from the browser are first broken into units of up to 16 KB. If compression is enabled, each unit is then separately compressed. After that, a secret key derived from the two nonces and premaster key is concatenated with the compressed text and the result hashed with the agreed-on hashing algorithm (usually MD5). This hash is appended to each fragment as the MAC. The compressed fragment plus MAC is then encrypted with the agreed-on symmetric encryption algorithm (usually by XORing it with the RC4 keystream). Finally, a fragment header is attached and the fragment is transmitted over the TCP connection.

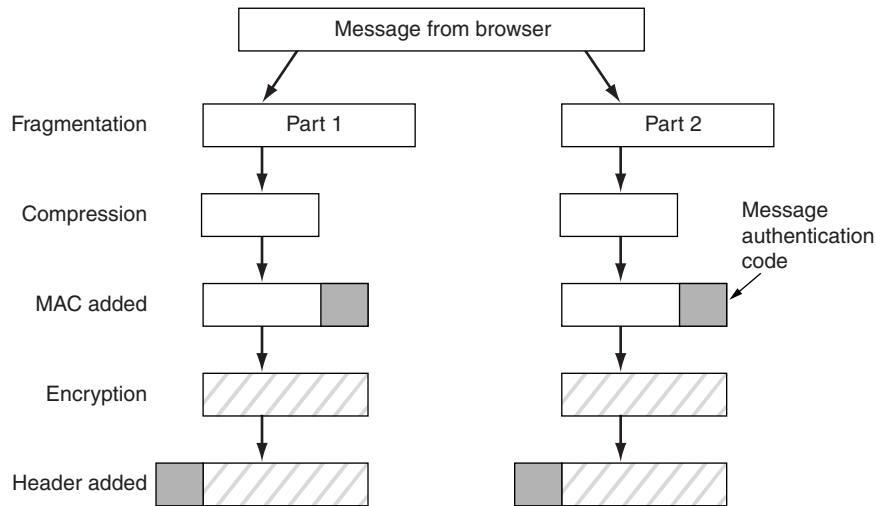


Figure 8-52. Data transmission using SSL.

A word of caution is in order, however. Since it has been shown that RC4 has some weak keys that can be easily cryptanalyzed, the security of SSL using RC4 is on shaky ground (Fluhrer et al., 2001). Browsers that allow the user to choose the cipher suite should be configured to use triple DES with 168-bit keys and SHA-1 all the time, even though this combination is slower than RC4 and MD5.

Another problem with SSL is that the principals may not have certificates and even if they do, they do not always verify that the keys being used match them.

In 1996, Netscape Communications Corp. turned SSL over to IETF for standardization. The result was **TLS (Transport Layer Security)**. It is described in RFC 2246.

The changes made to SSL were relatively small, but just enough that SSL version 3 and TLS cannot interoperate. For example, the way the session key is derived from the premaster key and nonces was changed to make the key stronger (i.e., harder to cryptanalyze). The TLS version is also known as SSL version 3.1. The first implementations appeared in 1999, but it is not clear yet whether TLS will replace SSL in practice, even though it is slightly stronger. The problem with weak RC4 keys remains, however.

8.9.4 Mobile Code Security

Naming and connections are two areas of concern related to Web security. But there are more. In the early days, when Web pages were just static HTML files, they did not contain executable code. Now they often contain small programs, including Java applets, ActiveX controls, and JavaScripts. Downloading

and executing such **mobile code** is obviously a massive security risk, so various methods have been devised to minimize it. We will now take a quick peek at some of the issues raised by mobile code and some approaches to dealing with it.

Java Applet Security

Java applets are small Java programs compiled to a stack-oriented machine language called **JVM (Java Virtual Machine)**. They can be placed on a Web page for downloading along with the page. After the page is loaded, the applets are inserted into a JVM interpreter inside the browser, as illustrated in Fig. 8-53.

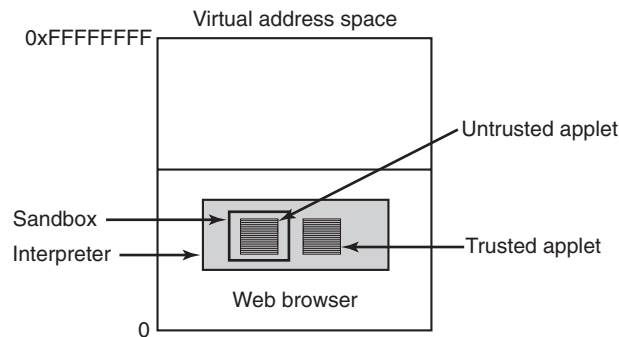


Figure 8-53. Applets can be interpreted by a Web browser.

The advantage of running interpreted code over compiled code is that every instruction is examined by the interpreter before being executed. This gives the interpreter the opportunity to check whether the instruction's address is valid. In addition, system calls are also caught and interpreted. How these calls are handled is a matter of the security policy. For example, if an applet is trusted (e.g., it came from the local disk), its system calls could be carried out without question. However, if an applet is not trusted (e.g., it came in over the Internet), it could be encapsulated in what is called a **sandbox** to restrict its behavior and trap its attempts to use system resources.

When an applet tries to use a system resource, its call is passed to a security monitor for approval. The monitor examines the call in light of the local security policy and then makes a decision to allow or reject it. In this way, it is possible to give applets access to some resources but not all. Unfortunately, the reality is that the security model works badly and that bugs in it crop up all the time.

ActiveX

ActiveX controls are Pentium binary programs that can be embedded in Web pages. When one of them is encountered, a check is made to see if it should be executed, and if it passes the test, it is executed. It is not interpreted or sandboxed

in any way, so it has as much power as any other user program and can potentially do great harm. Thus, all the security is in the decision whether to run the ActiveX control.

The method that Microsoft chose for making this decision is based on the idea of **code signing**. Each ActiveX control is accompanied by a digital signature—a hash of the code that is signed by its creator using public key cryptography. When an ActiveX control shows up, the browser first verifies the signature to make sure it has not been tampered with in transit. If the signature is correct, the browser then checks its internal tables to see if the program's creator is trusted or there is a chain of trust back to a trusted creator. If the creator is trusted, the program is executed; otherwise, it is not. The Microsoft system for verifying ActiveX controls is called **Authenticode**.

It is useful to contrast the Java and ActiveX approaches. With the Java approach, no attempt is made to determine who wrote the applet. Instead, a run-time interpreter makes sure it does not do things the machine owner has said applets may not do. In contrast, with code signing, there is no attempt to monitor the mobile code's run-time behavior. If it came from a trusted source and has not been modified in transit, it just runs. No attempt is made to see whether the code is malicious or not. If the original programmer *intended* the code to format the hard disk and then erase the flash ROM so the computer can never again be booted, and if the programmer has been certified as trusted, the code will be run and destroy the computer (unless ActiveX controls have been disabled in the browser).

Many people feel that trusting an unknown software company is scary. To demonstrate the problem, a programmer in Seattle formed a software company and got it certified as trustworthy, which is easy to do. He then wrote an ActiveX control that did a clean shutdown of the machine and distributed his ActiveX control widely. It shut down many machines, but they could just be rebooted, so no harm was done. He was just trying to expose the problem to the world. The official response was to revoke the certificate for this specific ActiveX control, which ended a short episode of acute embarrassment, but the underlying problem is still there for an evil programmer to exploit (Garfinkel with Spafford, 2002). Since there is no way to police thousands of software companies that might write mobile code, the technique of code signing is a disaster waiting to happen.

JavaScript

JavaScript does not have any formal security model, but it does have a long history of leaky implementations. Each vendor handles security in a different way. For example, Netscape Navigator version 2 used something akin to the Java model, but by version 4 that had been abandoned for a code signing model.

The fundamental problem is that letting foreign code run on your machine is asking for trouble. From a security standpoint, it is like inviting a burglar into

your house and then trying to watch him carefully so he cannot escape from the kitchen into the living room. If something unexpected happens and you are distracted for a moment, bad things can happen. The tension here is that mobile code allows flashy graphics and fast interaction, and many Web site designers think that this is much more important than security, especially when it is somebody else's machine at risk.

Viruses

Viruses are another form of mobile code. Only unlike the examples above, viruses are not invited in at all. The difference between a virus and ordinary mobile code is that viruses are written to reproduce themselves. When a virus arrives, either via a Web page, an e-mail attachment, or some other way, it usually starts out by infecting executable programs on the disk. When one of these programs is run, control is transferred to the virus, which usually tries to spread itself to other machines, for example, by e-mailing copies of itself to everyone in the victim's e-mail address book. Some viruses infect the boot sector of the hard disk, so when the machine is booted, the virus gets to run. Viruses have become a huge problem on the Internet and have caused billions of dollars worth of damage. There is no obvious solution. Perhaps a whole new generation of operating systems based on secure microkernels and tight compartmentalization of users, processes, and resources might help.