CHAPTER

Securing Sockets with SSL

23

What You Will Learn

In this chapter, you will learn about the secure sockets layer (SSL) and how it is used on Web sites. We investigate the layers and operation of the SSL protocol and discuss the SSL's use of certificates.

You will learn about the public key infrastructure (PKI) and how public keys are used for encryption. We present a simple example of public key encryption and decryption using only a pocket calculator and no advanced mathematics.

Web site security and user authentication were not much of a concern in the HTTP chapter. But the popularity of the Web for e-commerce is based on trusting that the transactions sent over the Internet are secure. To most users, this means two things:

Server authentication—The identity of the server is vouched for in some way (such as a *certificate*), so that users have confidence that the Web site is not run by a bunch of hackers collecting credit card or password information.

Safe passage—Data that passes back and forth between client and server cannot be read (decrypted) by hackers sniffing odd interfaces here and there.

In this chapter, we explore the SSL, the most widely deployed security protocol on the Web (and in the world) today. Many users notice the little yellow lock that appears in the lower right-hand corner of most Web browsers, and a large percentage of those realize that this means the browser has deemed this site "secure," but few bother to investigate just what that means.

SSL AND WEB SITES

In the last chapter, we configured the hosts `bsdserver` and `winsvr1` to act as a Web site using Apache. In this chapter, we'll explore the security aspects of the Web software. We'll be using the same equipment as in the previous chapter, as shown in Figure 23.1.

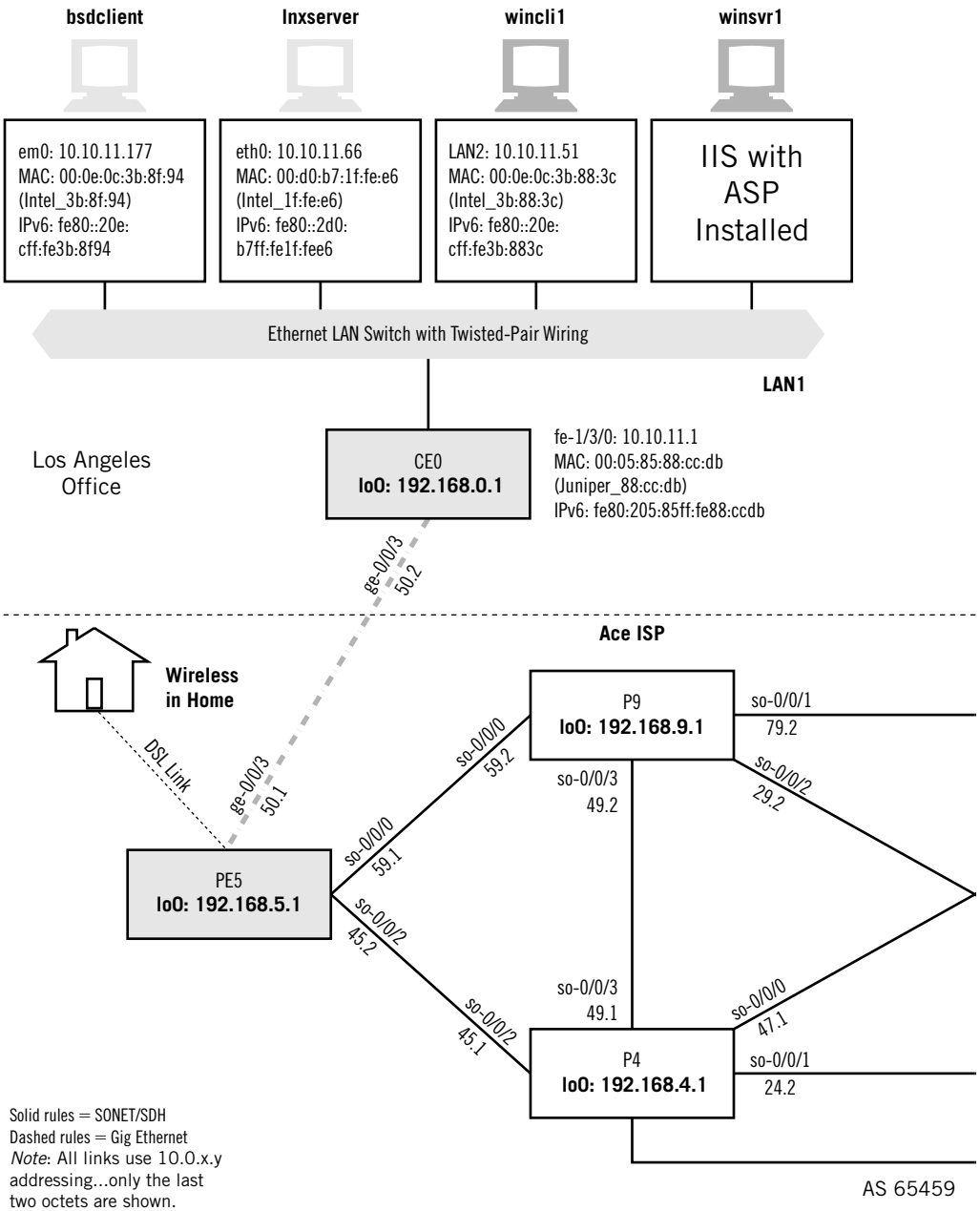
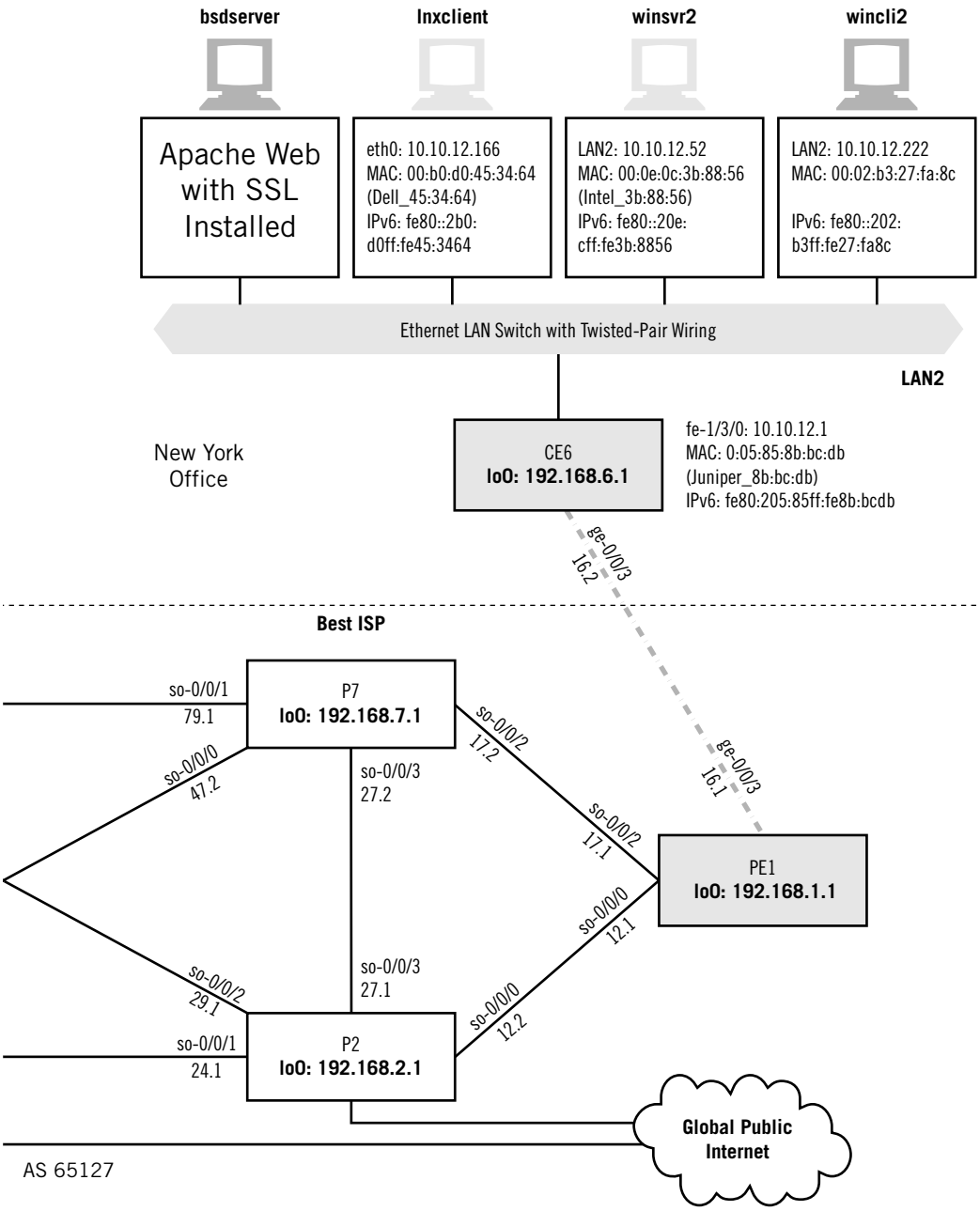


FIGURE 23.1

Web sites on the Illustrated Network showing that the Apache Web server supports SSL.


bsdserver
lnxclient
winsvr2
wincli2
Apache Web with SSL Installed
eth0: 10.10.12.166
MAC: 00:b0:d0:45:34:64
(Dell_45:34:64)
IPv6: fe80::2b0:
d0ff:fe45:3464
LAN2: 10.10.12.52
MAC: 00:0e:0c:3b:88:56
(Intel_3b:88:56)
IPv6: fe80::20e:
cff:fe3b:8856
LAN2: 10.10.12.222
MAC: 00:02:b3:27:fa:8c
IPv6: fe80::202:
b3ff:fe27:fa8c
Ethernet LAN Switch with Twisted-Pair Wiring
LAN2
New York Office
CE6
lo0: 192.168.6.1
fe-1/3/0: 10.10.12.1
MAC: 0:05:85:8b:bc:db
(Juniper_8b:bc:db)
IPv6: fe80:205:85ff:fe8b:bcdb
ge-0/0/3
16.2
Best ISP
so-0/0/1
79.1
P7
lo0: 192.168.7.1
so-0/0/2
17.2
so-0/0/0
47.2
so-0/0/3
27.2
ge-0/0/3
16.1
so-0/0/2
17.1
PE1
lo0: 192.168.1.1
so-0/0/0
12.1
so-0/0/2
29.1
so-0/0/3
27.1
so-0/0/0
12.2
so-0/0/1
24.1
P2
lo0: 192.168.2.1
Global Public Internet
AS 65127

The Apache Web server software uses a type of SSL called OpenSSL. What happens when we use the Apache Web server with the OpenSSL module on `bsdserver`? Let's try it from `wincli2` and see what happens. In the HTTP chapter, when we accessed the default Apache Web page (`index.html`) at *http://bsdserver.booklab.englab.jnpr.net*, the page mentioned SSL but did not display a security lock.

When we type in a request for the *secure* part of the `bsdserver` by using `https`, as in `https://bsdserver.booklab.englab.jnpr.net`, we get a default security alert right away from IE (as shown in Figure 23.2). It seems odd to warn about a secure connection, but that's what it does.

FIGURE 23.2

A security alert in IE, oddly "alerting" the user that the information *cannot* be viewed by others. Note that these warnings can be disabled.

FIGURE 23.3

A certificate security warning. Often the certificate has expired and has not yet been renewed.

Most people choose not to see this warning over and over and click the box, but it's good to see that the browser knows that it's going to establish a secure connection. If we okay the operation, the first thing that is noticeable is how much slower the server is to respond compared to the "regular" default Web page display—which is just about instantaneous because the two hosts are on the same LAN. Of course, the `bsdserver` is not the fastest platform, or the platform of choice, for commercial Web site hosting.

A lot is going on between server and client, but eventually the browser receives the *site certificate* and in this case immediately objects to the certificate provided by `bsdserver`. This is shown in Figure 23.3.

The certificate must pass three major tests, and the certificate used for testing OpenSSL with Apache is wanting in all three categories. First, the issuing "company" does not exist. Second, the certificate has expired. Third, the name on the certificate has nothing to do with `bsdserver`. The user can view the certificate, and ultimately decide to proceed or essentially abort the request for the page. If we view the certificate used for testing in Apache SSL, the reasons for the warnings become obvious (as shown in Figure 23.4).

The testing certificate issued by the nonexistent Snake Oil CA not only expired long ago but is issued to a bogus domain. Nevertheless, the user can choose to view the

FIGURE 23.4

Apache SSL test certificate, which fails on all three counts.

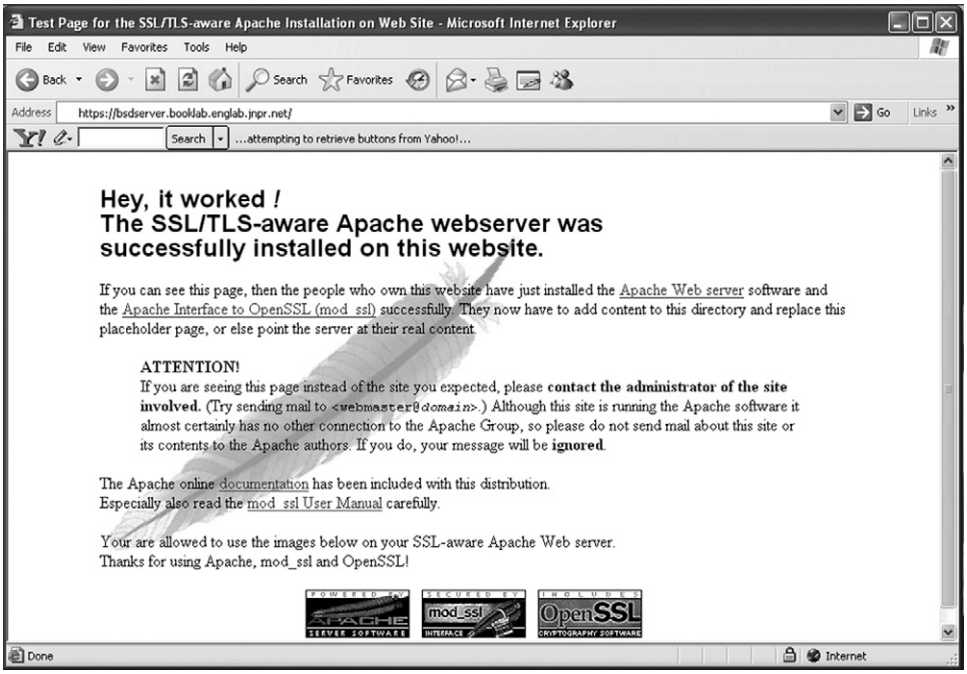


FIGURE 23.5

The secure Web page and lock (IE 7 moves it to the top of the page). Note the use of `https`.

details of the certificate fields, optionally store a copy of the certificate on the client, or choose to proceed (users cannot say they have not been warned!).

Clicking on `OK` finally (after another longish wait) delivers the secure Web page and displays the familiar browser secure lock in the lower right-hand corner of the window. We haven't actually installed any "real" secure pages, so the same page is used for content as in the last chapter. However, the content is sent encrypted to the client—which is the point. The page and lock are shown in Figure 23.5. IE7 moves the lock to the top of the page, but it's the same lock.

We can always view the certificate again by double-clicking on the lock. We see the same view as in Figure 23.4. The Details tab provides information about the certificate. The following are the fields in the Snake Oil certificate in detail.

- *Version*—V3 (SSLv3)
- *Serial Number*—01
- *Signature algorithm*—md5RSA
- *Issuer*—`ca@snakeoil.dom`, Snake Oil CA, Snake Oil, Ltd, Snake Town, Snake Desert, XY

- *Valid From*—Thursday, October 21, 1999 11:21:51 AM
- *Valid To*—Saturday, October 20, 2001 11:21:51 AM
- *Subject*—`www@snakeoil.dom`, `www.snakeoil.dom`, Webserver Team, Snake Oil, Ltd, Snake Town, Snake Desert, XY
- *Public key*—RSA (1024 bits; all 128 bytes follow)
- *Subject alternative name*—RFC822 Name=`www@snakeoil.dom`
- *Netscape comment*—mod ssl generated custom server certificate
- *Netscape Cert Type*—SSL Server Authentication (40)
- *Thumbprint algorithm*—sha1
- *Thumbprint*—20 bytes displayed

The Ethereal capture of the session shows that it takes 98 packets between client and server for an entire secure exchange. It also took almost 3 *minutes* to load the SSL page, but much of this time was "user think time" spent examining the warnings and alerts for the purposes of this book.

There is much more that could be explored in SSL, but the procedures become complex very quickly. Interested readers are referred to texts devoted to security issues. The rest of this chapter explores in more detail what we've just seen.

The Lock

The lock in the browser always gives users the strength of encryption used. Passing the mouse over the lock and pausing it will display a message box with text such as `SSL Secured (128 Bit)` in Internet Explorer (IE). This means that the keys used for encryption and decryption are 128 bits long, barely respectable today. Other browsers have other ways of revealing this information.

If you double-click on the lock, you'll be able to see the certificate information and purpose—which is usually to verify the identity of the server (remote computer). The information should also show the domain for which the certificate was issued (such as `www.example.com`), which should match the Web site. The issuer of the certificate is available, as well as the dates the certificate is valid.

Modern browsers have a built-in security feature that displays a warning message when you try to send information to a Web site that has a certificate "problem." The certificate could have expired, or the name on the certificate might not match the Web site. The user can choose to proceed, or not, or view the certificate itself.

Servers use the certificate to derive two keys, public and private. The public key is part of the digital certificate sent to the client browser. The public key is used to encrypt initial data sent to the server to set up session keys for the transaction. The reason the public key is not used throughout will be examined later in this chapter.

Some people get their own personal certificates and use them to secure a lot of what they do on the Internet, even protecting their email messages. Let's take a closer look at how SSL works as a protocol layer in TCP/IP.

Secure Socket Layer

The SSL protocol was invented as a way to secure Web sites, but the status of SSL as a protocol layer allows it to be used for any client–server transactions as long as they use TCP. SSL is the basis of a related method, Transport Layer Security (TLS), defined in RFC4346. Both form a complete socket layer sitting above TCP and UDP and add authentication (you are who you say you are), integrity (messages have not been changed between client-server pairs), and privacy (through encryption) to the Internet.

Figure 23.6 shows the relationship between SSL/TLS and the socket interface. SSL and TLS are so closely related that they both use the same well-known port. Many implementations of SSL support TLS. In fact, Ethereal often parses bits as "TLS" instead of the expected "SSL" in many places.

Typical SSL implementations on the Internet only authenticate the server. That is, SSL is used as the de facto standard way client users can be sure that when they log on to *www.mybank.com* the server is really an official entity of MyBank and not a phony Web site set up by hackers to entice users to send account, Social Security, PIN, or other information hackers always find useful. SSL used by a server is indicated by the little "lock" symbol that appears in the lower right-hand corner of most Web browsers.

TLS 1.0 can be considered an extension of SSL 3.0 to include the client side of the transaction. SSL is still used in the Netscape and Internet Explorer browsers, and in most Web server software. Not all Web pages need to be protected with SSL or TLS, and SSL can be used free for noncommercial use or licensed for commercial applications.

Why would a Web server need to authenticate and protect the client? Well, consider the liability of and bad publicity for MyBank if *www.mybank.com* accepted a request on the part of a fake client user who transferred someone's assets to an offshore account and closed the accounts? Today, many activities that could easily be done over the Internet require a phone call or fax or letter with signature (or several of these!) to protect the server from phony clients.

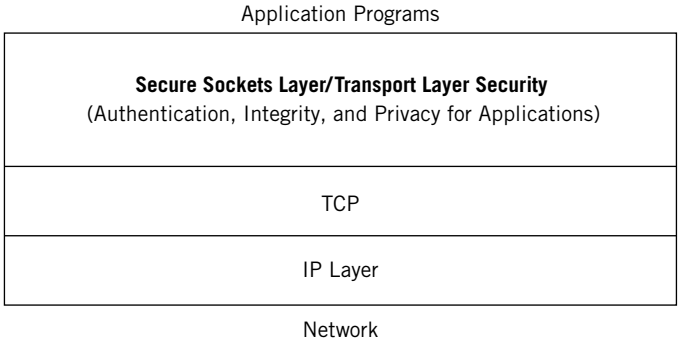


FIGURE 23.6

SSL/TLS as a "socket layer" protocol, showing how it sits on top of TCP.

PRIVACY, INTEGRITY, AND AUTHENTICATION

Before exploring SSL and TLS in more depth, an introduction to the methods they use to provide authentication, integrity, and privacy is necessary. A more complete discussion of these methods, especially certificates and public key cryptography, is presented in the chapter on IPSec.

Privacy

Privacy is the easiest for most to understand. Coded messages based on "conventional" or "traditional" secret keys have been used since ancient times, and anyone who has played with a "secret decoder ring" from a cereal box knows that the point is that only the sender and receiver know the *shared secret* key needed to code and decode the message. Most people also understand that such codes can be broken (some easily, some only with difficulty) by extensive analysis of the messages (the more text available, the better) or by simply finding out the "secret" key (the basis of many old spy movies). The key is the weakest point of the system: You can't use the code to protect the key for the same code because it is sent to other communication partners!

Today, public key (or asymmetrical) cryptography addresses the "key exchange problem" by using two keys—either one of which can be used to encrypt a message. One key remains private (i.e., known only to one party), whereas the other key is made public and available to anyone. Either key, public or private, can be used to encrypt a message—but then only the *other* key can be used to decrypt the message. (That's right, the key used for encryption can't even be used to "undo" the initial coding. Be careful when deleting the uncoded messages that the encrypted texts are based on!) A complete example of public key encryption is given later in this chapter.

Messages encrypted with the public key can only be decrypted by the private key, which means that the key exchange problem is solved. And if you give your public key to someone careless, it doesn't really matter: Anyone can learn the public key and the method is still secure as long as your private key remains private. Even better, we can now exchange old-fashioned shared secret keys this way and use them for a while (the longer a secret key is used, and the more text accumulates to analyze, the less secure the secret key). For instance, you can use your bank's public key to send transactions across the Internet and remain confident that only the bank can decrypt the message using its secret key.

Integrity

Traditional methods of making sure that the message sent is the one received left a lot to be desired. Witnessing documents with other signers, using public notaries, and other methods all had problems that could be circumvented. Traditional message integrity simply relied on the strength of the encryption method to make sure that no one "in the middle" had changed the message in transit. It is one thing to tell MyBank "transfer $10,000 to pay off my credit cards" and another to find out MyBank thought you said

"transfer $10,000 to Harry Hacker." As fascinating as your broken bank correspondence might be to read, hackers usually really want to do some damage. Then, as soon as the wire transfer has cleared, Harry can close his account and move on to the next victim.

Those who have been around networks know the concept of a frame *checksum* or *one-way hash*. The checksum is a fixed number of extra bits appended to a frame (message) to verify that no bits have been altered by errors on the network while the frame is in transit. Even the checksum itself is included in the "protection." The modern equivalent of the checksum hash, extended to many more bits and applied to the message text itself (or layers of the message plus headers added), is called a *message digest*. A message digest is just a big one-way hash, which means that the original text cannot be recovered from the hash value. On the other hand, the changes made *might* just yield the same hash value as the original message. Message digests understand this and are mathematically designed to make sure the chances of this happening are very slim, on the order of one chance in a million or better.

An associated use of message digests is as a *digital signature*. After all, the message digest hash only says that the message to MyBank arrived unaltered. It doesn't guarantee that the message really came from me. Anyone in the middle knowing the message digest algorithm can simply substitute the entire message, append the proper message digest, and sent it on to the bank.

But a digital signature involves more than just a hash on the message. A digital signature is used with public key encryption to encrypt not only the text and hash value but other information (such as a sequence number) with my private key. The digital signature is appended to the encrypted message and is valid *only* for that message. The digital signature can be decrypted with my public key, which might sound like defeating the purpose—but the point is that only you can create a digital signature using the message digest, and no one can change the digest and still sign it as you have (as long as my private key remains private, of course). No one else can use this signature later, for the same reason. Digital signatures provide the receivers with *nonrepudiation*, meaning that MyBank can be sure that you sent the message and that it's really the message you sent (again, as long as you protect your private key).

Authentication

There is only one more concept that remains in understanding how SSL and TLS work. This is the idea of a certificate. Thus far, we have developed a way for an individual to send encrypted, unalterable, signed messages to MyBank at *www.mybank.com*. We do this using the bank's public key, available to anyone. (Of course, the digital signature depends on the public key—although the certificate concept applies here as well.) But how do you know that the public key provided is really the bank's key? Where does MyBank's public key come from?

It comes from a certificate, of course. The bank provides me with a certificate confirming the public key and the identity of the holder of the key. How do you know the certificate is real? After all, all forms of encryption and authentication are susceptible to the "man-in-the-middle" exploit—where someone is busily intercepting messages

between client and server and substituting their own certificates (with their own keys) to both parties. One solution would be to hardcode the certificates into every browser, but this solution does not scale.

A more practical answer to the "man-in-the-middle" threat is that you know the certificate is real because you got it from a *certificate authority* (CA). The CA is a trusted third-party agency whose job it is to distribute certificates, usually on behalf of commercial enterprises that pay for their services. Certificates associate a public key with the identity of a *subject* (server or user), along with the public key. The CA issuer digital signature is included, as well as a period of validity (start and end), version and serial number of the certificate, and sometimes "extension" information.

CAs often require that certificate information be delivered in person by more than one validated representative of the company being "certified." This *root level* CA is also covered by a certificate, but one that is *self-signed*. Even on the Internet, someone has to be trusted implicitly. Other CAs can issue the certificate in a *certificate chain*. Some certification users refuse to accept a certificate if the chain is too long (the longer the chain, the greater the risk that one certificate in the chain might be bad).

Before central bank regulation became common, anyone could found a bank just by getting people to trust them with their money. Today, anyone can follow a few rules and be a CA and issue certificates—and that is especially true for private intranets in a large organization. Among the rules are procedures for validating, managing, and revoking certificates through *certificate revocation lists* (CRLs). CRLs are needed because certificates are passed around a lot and it is impossible to tell just by examination that a certificate is no longer valid because things have changed or it has been compromised or abused.

If the concepts of public key encryption, message digests, digital signatures, and certificates still seem somewhat vague and abstract, that's only to be expected. These are difficult concepts that take time to assimilate. The IPSec chapter revisits the concepts in more detail, and gives examples of how these concepts all work together.

PUBLIC KEY ENCRYPTION

Public key encryption, using a private key to recover what is encrypted with a public key, is based on complex mathematical principles. But that doesn't mean that the use of public key encryption is all that difficult to perform. After all, computers do it with ease.

Let's use something no more complex than an ordinary pocket calculator to perform this type of encryption. Along the way, several important points about public key encryption will be uncovered.

Pocket Calculator Encryption at the Client

The security that public key encryption provides is a consequence of the difficulty of factoring large numbers, not the complexity of the method. You can do PKI on any pocket calculator. The "how" is shown in the "Three Magic Numbers" sidebar and explained in material following.

Three Magic Numbers

1. Start with three magic numbers: Public "normalizer" N = 33, public encryption key E = 3, and private decryption key D = 7.
2. Encrypt plain-text letter "O" (15th letter of the alphabet) from certificate N and E values.
3. Write down "O" value E times and multiply:
   $15 \times 15 \times 15 = 3375$
4. Divide by N and compute remainder:
   $3375/33 = 102.27272\ldots$
   $0.27272\ldots \times 33 = 8.99976 = \mathbf{9}$
5. Send **9**, the cipher text for plain-text 15, over the network.

We have to start with three "magic" numbers, and two of them must be prime numbers. Usually, you choose two large primes first (hundreds of digits) and derive a third huge number called N (for "normalizer") through a very complex process. N is never called a key in the documentation, but N is necessary for both encrypting and decrypting. The security comes from the fact that given a large N and one of the keys, it is next to impossible to derive the second prime key number. In this example, N = 33, and the two primes are 3 and 7. There is no obvious relationship between 33 and 3 and 7, although with these small numbers, a code cracker could figure it out in a minute or two.

One of the two primes becomes the public key (it doesn't matter which), and the other becomes the private key. Never consistently assign the smaller number as the public key. This speeds up client encryption, but is a security risk if people know one factor must be larger than the other. In this example, N = 33, the public encryption key E = 3, and the private decryption key D = 7.

Example

To encrypt the plain-text letter "O," first convert it to a number. "O" is the 15th letter of the alphabet; we can use that. Of course, we have to obtain the values of the server's N and E values. We can get those from a certificate, in that the values of N and E must match up properly with the D that the receiver retains.

Now write down the "O" value E times and multiply, using any suitable calculator with at least eight (8) positions. So, $15 \times 15 \times 15 = 3375$. This is not too large, so the encryption does not need N yet.

Divide by N and compute remainder. This is just $3375/33 = 102.27272$. The fraction is there because calculators do not give remainders directly. We can get it by subtracting 102, leaving 0.27272. Then, $0.27272 \times 33 = 8.99976 = \mathbf{9}$. We have to round a little due to the limited precision of the decimal fraction. The client sends **9**, which is the cipher text for the 15 ("O") plain text, over the network.

At the Server

1. Get back "O" without using E, but only N = 33 and D = 7. The receiver gets cipher-text **9** over the network.
2. Write down cipher-text value D (7) times and multiply, applying "normalizer" whenever number gets large:
   $9 \times 9 \times 9 \times 9 \times 9 \times 9 \times 9 = (531{,}441) \times 9$
   But 531,441/33 = 16,104.272 and $0.272 \times 33 = 8.976 = 9$.
   So, $(9) \times 9 = 81$.
   Divide the final result by N and compute the remainder:
   $81/33 = 2.4545454\ldots$
   $0.4545454 \times 33 = 14.99998 =$ **15**
3. Thus, **15** plain text is the letter "O" sent securely.

Pocket Calculator Decryption at the Server

Thus far, the client has used the proper N and E from the server to encrypt "O" (15) as cipher-text **9**. This is what is sent on the network. The magic of PKI is being able to get back "O" without using E, only N and D. (Because N is known to and used by both parties, it is never called a key itself.) In this example, N = 33, E = 3, and D = 7. The following is how to get back "P" using only N = 33 and D = 7 at the server end.

1. Write down the cipher-text value (**9**) D times and multiply. If the number gets too large for the calculator, we can apply N to get back a more useable number.
   9 3 9 3 9 3 9 3 9 3 9 3 9 5 (531,441) 3 9
   If we don't want to risk overflowing the calculator, we can apply N at any time as follows:
   531,441/33 = 16,104.272 (subtract 16,104) and $0.272 \times 33 = 8.976 = 9$
   (Again, rounding is needed to deal with the annoying decimal fractions that calculators insist on providing.)
   So, $(9) \times 9 = 81$. Note how the single (9) replaces 531,441. It is just a coincidence that this turned out to be 9 also.
2. Divide the final result by N and compute remainder:
   81/33 5 2.4545454, so subtract 2
   $0.4545454 \times 33 = 14.99998 = 15$

3. Thus, the plain-text 15 is the letter "O" sent securely using PKI. That's all there is to it! Of course, usually it's a number that's encrypted—but so what? Try the number 19 for yourself. You might have to "normalize" on the encryption side as well, but it still works.

The security in PKI is in the difficulty of finding D given the values of E and N. This example is mathematically trivial to hackers and crackers. But try N = 49,048,499 and E = 61. The answer is D = 2,409,781. Usually, N, E, and D are anywhere from 140 to 156 or more digits long. To deal with text messages, strings of letters can be thought of as numbers. So, "OK" becomes 1511. ASCII is typically used.

Digital signatures employ the same public keys as well. Either key, E or D, can be used to encrypt or decrypt. You just need to use the other to reverse the process (try it with "O"). So, any message *encrypted* with D can only be *decrypted* with E (my public key). So, any text that can be decrypted with E (and N) *had* to come from me as long as my private key D remains secure.

PUBLIC KEYS AND SYMMETRICAL ENCRYPTION

As has just been pointed out, public key encryption is done routinely by computers—but it's not an easy task, even for modern processors. Computers are really an engineering tool and were generally scorned by mathematicians until relatively recently. In fact, sometimes a mathematician will ask a computer scientist what value of π is used in computations. Any value that contains less than an infinite number of digits is incorrect, of course. At some point the loss of accuracy is fine for engineers, but not for "pure" mathematicians.

So, the length of the strings encrypted with public keys must be limited to what a computer can handle. We have to admit, the first time we heard about "128-bit encryption," we thought it would be interesting because no programming languages at the time supported "integers" longer than 64 bits—let alone powers involving 128-bit numbers. Normalization helps, of course, but the computational drain of public keys on general processors is substantial.

For this reason, SSL uses public key encryption as little as possible—typically only to establish symmetrical keys that can be used much more efficiently with existing algorithms and processors. Naturally, the symmetrical keys are *much* less secure than public key encryption, but they are changed more often and used for shorter periods of time.

SSL AS A PROTOCOL

SSL is a protocol layer all on its own that is placed between a connection-oriented, network layer protocol (almost always TCP) and the application layer protocol (such as HTTP) or program. Connections are useful to provide a convenient way to

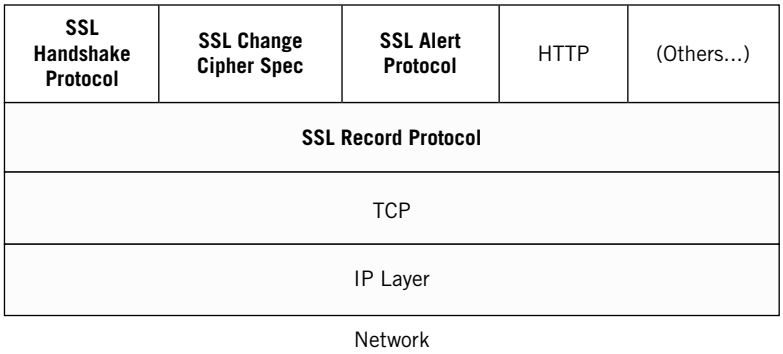


FIGURE 23.7

The SSL protocol stack in detail showing its relationship to HTTP and other protocols.

associate security parameters with a specific flow of packets. SSL uses certificates for authentication, digital signatures and message digests for integrity, and encryption for privacy. Each of the three security areas has a range of choices allowed in order to respect local laws regarding cryptographic algorithms and new technologies to be included as developed. Specific choices in each area are negotiated when a protocol session (connection) is set up.

SSL Protocol Stack

The SSL protocol stack is shown in Figure 23.7. TLS can be regarded as an enhanced version of the SSL protocol stack, but the components are essentially the same.

SSL usually uses Diffie-Hellman (a secure key exchange method used on unsecure networks) to exchange the keys. The handshake procedure itself uses three SSL protocol processes: the *SSL Handshake Protocol* for the overall process, the *SSL Change Cipher Spec Protocol* for Cipher Suite specification and negotiation, and the *SSL Alert Protocol* for error messages.

All three of these protocols use the *SSL Record Protocol* to encapsulate their messages, as well as the application data flowing on the session once established. The nice thing about the SSL Record Protocol is that it provides a way to renegotiate active session parameters or establish a new session using a secure path. Initial session handshakes without a functioning and secure SSL Record Protocol must use a *NULL Cipher Suite* (plain text), which is of course a risk.

SSL Session Establishment

Established SSL sessions can be reused, which is good because the SSL session establishment process requires the exchange of many messages. Sessions are established after a complex handshake routine between client and server. There are many

variations in the details of SSL session establishment, but Figure 23.8 shows one of the most common.

By default, SSL uses TCP port 443. Of course, a user typically just uses `http://` (or nothing at all) when accessing a Web page. Rather than making users remember to type in the port number at the end of the URL, SSL is invoked with a URL starting with `https://`. This should not be confused with Web pages distinguished by the `.shtml` ending, which means that the Server Side Includes (SSIs) are in use for that page. There are four major phases to the SSL session establishment process.

1. Initial Hello exchange
2. Optional server certificate presentation and request (authentication of server to client)
3. Presentation of client certificate if requested (authentication of client to server)
4. Finalize Cipher Suite negotiation and finish session establishment handshake

Usually, only the server presents its certificate to the client (user). Most users don't have certificates to authenticate themselves to the server, but this will change with TLS. Regarding Cipher Suite negotiation, SSL 3.0 defines 31 Cipher Suites consisting of a key exchange method, the cipher (encryption method) to use for data transfer, and the

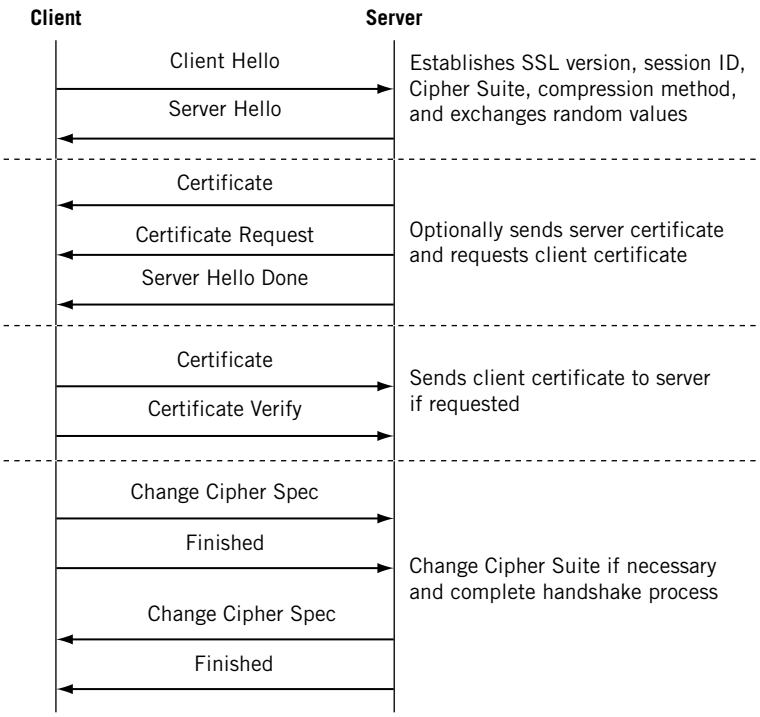


FIGURE 23.8

One form of SSL session establishment. There can be others, but this form is very common.

message digest method to use to create the SSL *Message Authentication Code* (MAC). There are nine choices for the traditional shared secret key encryption used in SSL.

- No encryption
- 40-bit key RSA Data Security, Inc. Code (RC4) stream cipher
- 128-bit key RC4 stream cipher
- 40-bit key RC2 Cipher Block Chaining (CBC)
- The venerable Data Encryption Standard (DES), DES40, and Triple DES (3DES), all with CBC
- Idea
- Fortezza

CBC uses a portion of the previously encrypted cipher text to encrypt the next block of text. There are three choices of message digest.

- No message digest
- 128-bit hash Message Digest 5 (MD5)
- 160-bit hash Secure Hash Algorithm (SHA)

SSL Data Transfer

All application data and SSL control data use the SSL Record Protocol for message transfer. Details vary, but usually the SSL Record Protocol will fragment the application data stream (perhaps a Web page) into record protocol units. Each unit is typically compressed (compression adds a layer of complexity to unauthorized decryption attempts), and the MAC is computed before the entire unit is encrypted. The end result is tucked into a TCP segment and IP packet and sent on its way. This process is illustrated in Figure 23.9.

SSL Implementation

Few programmers write an SSL implementation from scratch. SSL is usually implemented as a *toolkit library*, and patented cryptographic functions must be licensed anyway. Public key packages are patented as well, and there are export restrictions on cryptographic algorithms in the United States. All of these factors combine to discourage individuals from implementing SSL (as opposed to plain sockets) on their own.

Two public key toolkits are popular. RSARef is the RSA "reference" public key package, including RSA encryption and Diffie-Hellman key exchange. It also features unsupported, but free, source code and is to be used for noncommercial applications. BSAFE3.0 ("Be-safe," not an acronym) is the commercial version of RSARef. The public key toolkits can be combined with any SSL toolkits, including:

SSLRef—An example SSL 3.0 implementation from Netscape Communications Corp.

SSLava—An SSL 3.0 toolkit from Phaos Technology written in Java.

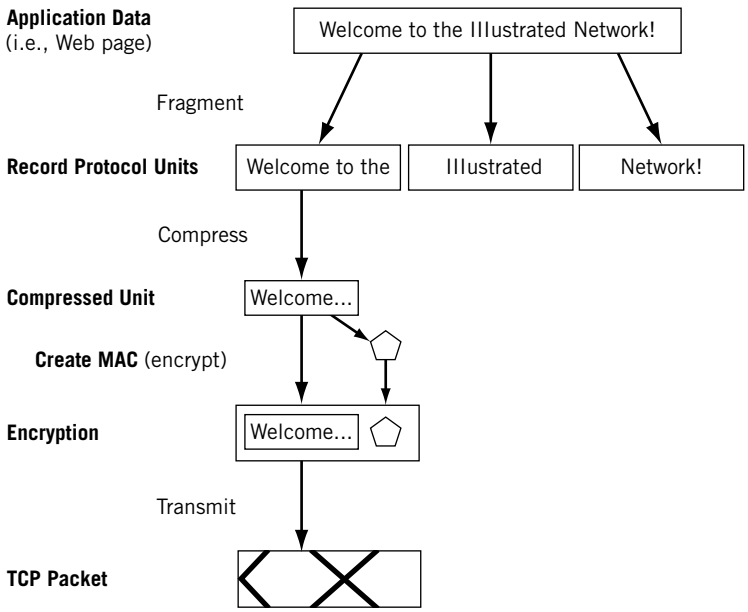


FIGURE 23.9

The SSL record protocol showing how protocol units are compressed and encrypted.

OpenSSL—A free noncommercial implementation of SSL 3.0 (and 2.0) and TLS 2.0) that can be used outside the United States. In the United States, patent restrictions require use of RSARef or BSAFE3.0.

SSL Issues and Problems

SSL is not perfect, of course. SSL suffers from a number of limitations, most of which can be overcome with careful planning and attention to detail. The sections that follow discuss a representative list of SSL issues.

Computational Complexity

As we've seen, public key encryption is so processor intensive that we avoid it whenever we can. And because the server must perform the SSL handshake for every connection, OpenSSL struggles under heavy workloads. Hardware acceleration with special cards helps, and load balancing among multiple servers all representing the same Web site helps as well.

Clear Private Keys

The server has to store the private key somewhere, and usually in clear form (otherwise, we just move the issue to the next key, or the next, and restarts become a real problem unless the actual key is somewhere on the system). The point is, of course, that data

might be *transmitted* over the network in encrypted form but it is seldom *stored* on the server in an encrypted form. The physical security of the server is essential, and a technique called *perfect forward secrecy* is also helpful. We'll meet forward secrecy again in a discussion of IPSec.

Stolen Credentials

Certificate revocation lists are fine, but if a private key or certificate is stolen it can take a while for the organization to figure out that there is a bogus *www.example.com* site out there stealing people's money and identities. It's better to query the CA with a special protocol, such as the Online Certificate Status Protocol (OCSP)—defined in RFC 2560—but that's not common (and may never be). Again physical security is of paramount importance.

Pseudorandom Numbers and "Entropy"

In SSL, clients and servers both have to generate random numbers and data to use for session keys. The problem is that most computers' *pseudorandom number generators* (PRNGs) are not adequate for true security because they are predictable (one of the reasons they are *pseudo*random in the first place). The *seed* number used as input to the PRNG must itself be as random as possible, and many SSL implementations use seeds that do not have enough "entropy" (a measure of disorder or randomness). There are software-based workarounds for this.

Works Only with TCP

SSL only protects applications that use TCP. This is fine for HTTP, but more and more critical data on the Internet uses UDP and not TCP. We've already noted that multicast uses UDP, and we'll see that VoIP does as well. These data streams need protection, but SSL cannot currently provide it.

Inadequate Nonrepudiation

Suppose you purchase a product over the Internet that has a rebate. You have to send proof that you are the person that purchased the product to the rebate "fulfillment center" to receive the rebate. This is nonrepudiation in the sense that the company cannot say to the rebate center you didn't purchase the product. However, SSL cannot provide this nonrepudiation. The workaround, which involves the company *and* you having certificates, is relatively easy (but this will take a while to become the standard).

When using any security method, all of the system's "vulnerabilities" are difficult to seal. It's just difficult to detect and patch up all cracks in a complex system.

I once worked in an organization with a coworker who was famous for "playing" with the servers and their users by simply intercepting messages on the LAN. When the organization switched to encrypted communications, I tried to console him, thinking his hacking days were over. "That's all right," he told me, "I know where the backups are. Those aren't encrypted."

Where *are* those frequent backups of the Web servers' information? How secure are *they*? Security is always a never-ending battle where one side or the other seems to gain an advantage for a while, but never for long. Many of the limitations of SSL are

addressed in TLS 1.1, but TLS is new and most clients are not as sophisticated as servers when it comes to security.

A Note on TLS 1.1

The biggest shortcoming of SSL is the fact that as typically implemented only the server is authenticated to the user. That is, the server certificate with the server's public key and other information is presented to the client. But clients such as Web browsers seldom have certificates to present to the server to authenticate the user. Server authentication is fine for Internet commerce (encrypted personal and credit card information is sent to the server) but not so good for on-line banking and other applications where mutual authentication is desired, if not indispensable.

Implementation of TLS 1.1 (RFC4346) allows clients (users) to use the full capabilities of the standardized PKI. This topic is explored more fully in the chapter on IPSec.

SSL and Certificates

Let's take a close look at how SSL handles certificates. Ordinarily, once SSL is installed on a server you have to generate a certificate request to one of the major CAs (such as VeriSign). There are many types of certificates available, such as personal (mainly for email), code signing (for downloaded programs), and Web site (which is what we're talking about here).

Of course, the certificate has to be distributed by a CA, which also has to be set up. In OpenSSL, most CA operations can be done at the CLI, but this method is not really suitable for a production environment.

No matter which SSL server software is used, they all tell you how to generate a certificate signing request (CSR). Once this is done, the software generates a public/private key pair. You send the public key and the CSR to the certificate-issuing authority.

If all is in order when reviewed, including related documentation, the response is emailed to the applicant and loaded into the server SSL software. You usually get three things in the response:

- The CA's certificate containing the public key
- The local certificate identifying the server
- A certificate revocation list with a list of certificates revoked by the CA

For testing purposes, it is not necessary in most cases to obtain a "real" certificate. OpenSSL, for example, includes the testing certificate from the Snake Oil CA that is functional but not intended for use (hopefully, the "snake oil" name, used for useless tonics or medications, will be a tip-off to users).

QUESTIONS FOR READERS

Figure 23.10 shows some of the concepts discussed in this chapter and can be used to answer the following questions.

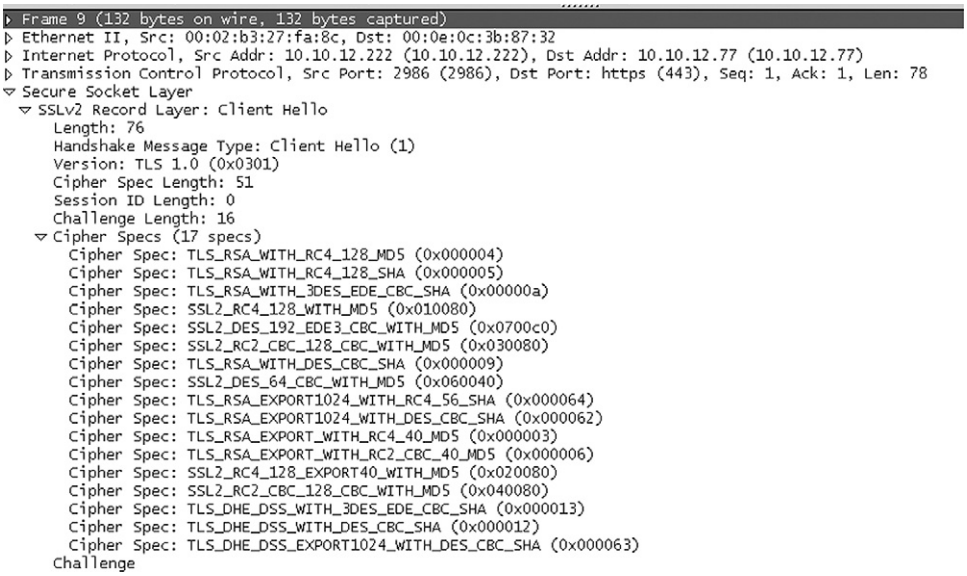


FIGURE 23.10

Ethereal capture of an SSL Client Hello frame. Note the list of encryption methods and details in the cipher suite.

1. Which port is used by `https`?
2. Which version of SSL is used at the record layer?
3. The capture says the "version" of SSL used is TLS 1.0. Why is that?
4. Which message should be sent in response to a Client Hello?
5. Is SSLv2 DES encryption with SHA supported by the client?