HOUR 17

Setting Up a Secure Server

This hour explains how to set up an Apache server capable of secure transactions. In this hour, you will learn

- The installation and configuration of the `mod_ssl` Apache module
- The SSL/TLS family of protocols and the underlying cryptography concepts
- What certificates are and how to create and manage them

The Need for Security

As the Internet became mainstream and the number of companies, individuals, and government agencies using it grew, so did the number and type of transactions that needed protection. Those include financial transactions, such as banking operations and electronic commerce, as well as exchange of sensitive information, such as medical records and corporate documents.

There are three requirements to carry on secure communications on the Internet: confidentiality, integrity, and authentication.

Confidentiality

Confidentiality is the most obvious requirement for secure communications. If you are transmitting or accessing sensitive information such as your credit card number or your personal medical history, you certainly do not want a stranger to get hold of it.

Integrity

The information contained in the exchanged messages must be protected from external manipulation. That is, if you place an order online to buy 100 shares of stock, you do not want to allow anyone to intercept the message, change it to an order to buy 1000 shares, or replace the original message. Additionally, you want to prevent an attacker from performing replay attacks, which, instead of modifying the original message, simply resend it several times to achieve a cumulative effect.

Authentication

You need to decide whether to trust the organization or individual you are communicating with. To achieve this, you must authenticate the identity of the other party in the communication.

The science of cryptography studies the algorithms and methods used to securely transmit messages, ensuring the goals of confidentiality, integrity, and authenticity. Cryptanalysis is the science of breaking cryptographic systems.

The SSL Protocol

SSL stands for Secure Sockets Layer and TLS stands for Transport Layer Security. They are a family of protocols that were originally designed to provide security for HTTP transactions, but that also can be used for a variety of other Internet protocols such as IMAP and NNTP. HTTP running over SSL is referred to as *secure HTTP*.

Netscape released SSL version 2 in 1994 and SSL version 3 in 1995. TLS is an IETF standard designed to standardize SSL as an Internet protocol. It is just a modification of SSL version 3 with a small number of added features and minor cleanups. The TLS acronym is the result of arguments between Microsoft and Netscape over the naming of the protocol because each company proposed its own name. However, the name has not stuck and most people refer to these protocols simply as SSL. Unless otherwise specified, the rest of this hour refers to SSL/TLS as *SSL*.

You specify that you want to connect to a server using SSL by replacing `http` with `https` in the protocol component of a URI. The default port for HTTP over SSL is 443.

The following sections explain how SSL addresses the confidentiality, integrity, and authentication requirements outlined in the previous section. In doing so, it explains, in a simplified manner, the underlying mathematical and cryptographic principles SSL is based on.

Confidentiality

The SSL protocol protects data from eavesdropping by encrypting it. Encryption is the process of converting a message, the *plaintext*, into a new encrypted message, the *ciphertext*. Although the plaintext is readable by everyone, the ciphertext will be completely unintelligible to an eavesdropper. Decryption is the reverse process, which transforms the ciphertext into the original plaintext.

Usually encryption and decryption processes involve an additional piece of information: a *key*. If both sender and receiver share the same key, the process is referred to as *symmetric* cryptography. If sender and receiver have different, complementary keys, the process is called *asymmetric* or *public key* cryptography.

Symmetric Cryptography

If the key used to both encrypt and decrypt the message is the same, the process is known as symmetric cryptography. DES, Triple-DES, RC4, and RC2 are algorithms used for symmetric key cryptography. Many of these algorithms can have different key sizes, measured in bits. In general, given an algorithm, the greater the number of bits in the key, the more secure the algorithm is and the slower it will run because of the increased computational needs of performing the algorithm.

Symmetric cryptography is relatively fast compared to public key cryptography, which is explained in the next section. Symmetric cryptography has two main drawbacks, however. One drawback is that keys should be changed periodically, to avoid providing an eavesdropper with access to large amounts of material encrypted with the same key. The other drawback is the key distribution problem: How to get the keys to each one of the parties in a safe manner? This was one of the original limiting factors, and before the invention of public key cryptography, the problem was solved by periodically having people traveling around with suitcases full of keys.

Public Key Cryptography

Public key cryptography takes a different approach. Instead of both parties sharing the same key, there is a pair of keys: one public and the other private. The public key can be widely distributed, whereas the owner keeps the private key secret. These two keys are

complementary; a message encrypted with one of the keys can be decrypted only by the other key.

Anyone wanting to transmit a secure message to you can encrypt the message using your public key, assured that only the owner of the private key—you—can decrypt it. Even if the attacker has access to the public key, he cannot decrypt the communication. In fact, you want the public key to be as widely available as possible. Public key cryptography can also be used to provide message integrity and authentication. RSA is the most popular public key algorithm.

The assertion that only the owner of the private key can decrypt it means that with the current knowledge of cryptography and availability of computing power, an attacker will not be able to break the encryption by brute force alone in a reasonable timeframe. If the algorithm or its implementation is flawed, realistic attacks are possible.

Public key cryptography is similar to giving away many identical lockpads and retaining the key that opens them all. Anybody who wants to send you a message privately can do so by putting it in a safe and locking it with one of those lockpads (public keys) before sending it to you. Only you have the appropriate key (private key) to open that lockpad (decrypt the message).

The SSL protocol uses public key cryptography in an initial handshake phase to securely exchange symmetric keys that can then be used to encrypt the communication.

Integrity

Data integrity can be preserved by performing a special calculation on the contents of the message and storing the result with the message itself. When the message arrives at its destination, the recipient can perform the same calculation and compare the results. If the contents of the message changed, the results of the calculation will be different.

Digest algorithms perform just that process, creating message digests. A *message digest* is a method of creating a fixed-length representation of an arbitrary message that uniquely identifies it. You can think of it as the fingerprint of the message. A good message digest algorithm should be irreversible and collision resistant, at least for practical purposes. *Irreversible* means that the original message cannot be obtained from the digest and *collision resistant* means that no two different messages should have the same digest. Examples of digest algorithms are MD5 and SHA.

Message digests alone, however, do not guarantee the integrity of the message because an attacker could change the text *and* the message digest. Message authentication codes, or MACs, are similar to message digests, but incorporate a shared secret key in the process. The result of the algorithm depends both on the message and the key used. Because the attacker has no access to the key, he cannot modify both the message and the digest. HMAC is an example of a message authentication code algorithm.

The SSL protocol uses MAC codes to avoid replay attacks and to assure integrity of the transmitted information.

Authentication

SSL uses certificates to authenticate parties in a communication. Public key cryptography can be used to digitally sign messages. In fact, just by encrypting a message with your secret key, the receiver can guarantee it came from you. Other digital signature algorithms involve first calculating a digest of the message and then signing the digest.

You can tell that the person who created that public and private key pair is the one sending the message. But how can you tie that key to a person or organization that you can trust in the real world? Otherwise, an attacker could impersonate his identity and distribute a different public key, claiming it is the legitimate one. Trust can be achieved by using digital certificates. *Digital certificates* are electronic documents that contain a public key and information about its owner (name, address, and so on). To be useful, the certificate must be signed by a trusted third party (certification authority, or CA) who certifies that the information is correct. There are many different kinds of CAs, as described later in the hour. Some of them are commercial entities, providing certification services to companies conducting business over the Internet. Other CAs are created by companies providing internal certification services.

The CA guarantees that the information in the certificate is correct and that the key belongs to that individual or organization. Certificates have a period of validity and can expire or be revoked. Certificates can be chained so that the certification process can be delegated. For example, a trusted entity can certify companies, which in turn can take care of certifying its own employees.

If this whole process is to be effective and trusted, the certificate authority must require appropriate proof of identity from individuals and organizations before it issues a certificate.

By default, browsers include a collection of root certificates for trusted certificate authorities.

SSL and Certificates

The main standard defining certificates is X.509, adapted for Internet usage. An X.509 certificate contains the following information:

- **Issuer:** The name of the signer of the certificate
- **Subject:** The person holding the key being certified
- **Subject public key:** The public key of the subject
- **Control information:** Data such as the dates in which the certificate is valid
- **Signature:** The signature that covers the previous data

You can check a real-life certificate by connecting to a secure server with your browser. If the connection has been successful, a little padlock icon or another visual clue will be added to the status bar of your browser. With Internet Explorer, you can click the locked padlock icon to open a page containing information on the SSL connection and the remote server certificate. You can access the same information by selecting Properties, and then Certificates from the File menu. Other browsers, such as Netscape, Mozilla, and Konqueror provide a similar interface.

Open the `https://www.ibm.com` URL in your browser and analyze the certificate, following the steps outlined in the preceding paragraph. You can see how the issuer of the certificate is the Equifax Secure E-Business Certification Authority-2, which, in turn, has been certified by the Thawte CA. The page downloaded seamlessly because Thawte is a trusted CA that has its own certificates bundled with Internet Explorer and Netscape Navigator.

To check which certificates are bundled with your Internet Explorer browser, select Tools, Internet Options, Content, Certificates, Trusted Root Certification Authorities.

You can see that both issuer and subject are provided as distinguished names (DN), a structured way of providing a unique identifier for every element on the network. In the case of the IBM certificate, the DN is C=US, S=New York, L=Armonk, O=IBM, CN=`www.ibm.com`.

C stands for country, S for state, L for locality, O for organization, and CN for common name. In the case of a Web site certificate, the common name identifies the fully qualified domain name of the Web site (FQDN). This is the server name part of the URL; in this case, `www.ibm.com`. If this does not match what you typed in the top bar, the browser will issue an error.

Figure 17.1 shows the certificate information described earlier.

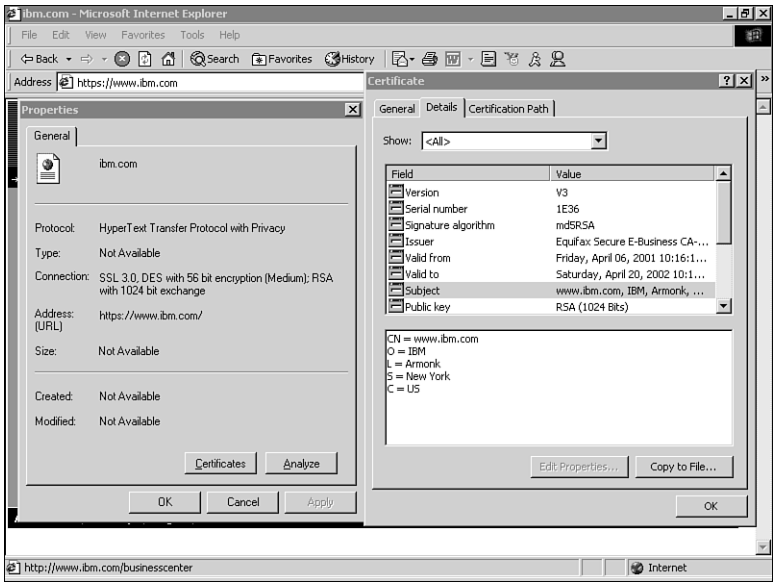

FIGURE 17.1
Certificate information.

17

SSL Protocol Summary

You have seen how SSL achieves confidentiality via encryption, integrity via message authentication codes, and authentication via certificates and digital signatures.

The process to establish an SSL connection is the following:

1. The user uses his browser to connect to the remote Apache server.
2. The handshake phase starts, and the browser and server exchange keys and certificate information.
3. The browser checks the validity of the server certificate, including that it has not expired, that it has been issued by a trusted CA, and so on.
4. Optionally, the server can require the client to present a valid certificate as well.
5. Server and client use each other's public key to securely agree on a symmetric key.
6. The handshake phase concludes and transmission continues using symmetric cryptography.

Installing SSL

Now that you've learned all about SSL, you need to install SLL support for Apache. SSL support is provided by `mod_ssl`, a module that is included with Apache but is not enabled

by default. `mod_ssl`, in turn, requires the OpenSSL library—an open source implementation of the SSL/TLS protocols and a variety of other cryptographic algorithms. OpenSSL is based on the SSLeay library developed by Eric A. Young and Tim J. Hudson. You can learn more about `mod_ssl` and OpenSSL in the Web sites noted in the reference section at the end of the hour.

OpenSSL

This section explains how to download and install the OpenSSL toolkit for both Windows and Unix variants.

Windows

At the time of writing this book, the Apache Software Foundation does not provide an SSL-enabled binary installer for Windows due to legal restrictions. That situation is likely to change soon, and you will be able to access precompiled SSL module and libraries. Check the Apache site for up-to-date information. The rest of the hour assumes that you have access to the `openssl.exe` command line utility, which will be included in the `bin/` directory of the SSL-enabled Apache distribution. It is a utility for generating certificates, keys, signing requests, and so on.

Unix

If you are running a recent Linux or FreeBSD distribution, OpenSSL might already be installed in your system. Use the package management tools bundled with your distribution to determine whether that is the case or, otherwise, to install it.

If you need to install OpenSSL from source, you can download OpenSSL from `http://www.openssl.org`. After you have downloaded the software, you need to uncompress it and `cd` into the created directory:

```
# gunzip < openssl*.tar.gz | tar xvf -
# cd openssl*
```

OpenSSL contains a `config` script to help you build the software. You must provide the path to which the software will install. The path used in this hour is `/usr/local/ssl/install`, and you probably need to have root privileges to install the software there. You can install the software as a regular user, but to do so, you will need to change the path. Then you must build and install the software:

```
# ./config --prefix=/usr/local/ssl/install \
--openssldir=/usr/local/ssl/install/openssl
# make
# make install
```

If everything went well, you have now successfully installed the OpenSSL toolkit. The `openssl` command-line tool will be located in `/usr/local/ssl/install/bin/`.

This tool is used to create and manipulate certificates and keys and its usage is described in a later section on certificates.

mod_ssl

In the past, SSL extensions for Apache had to be distributed separately because of export restrictions. Although there are limitations in redistribution of binaries that need to be solved and clarified, these restrictions no longer exist for distribution of source code, and mod_ssl is bundled and integrated with Apache 2.0. This section describes the steps necessary to build and install this module. `mod_ssl` depends on the OpenSSL library, so a valid OpenSSL installation is required.

Unix

If you are using the Apache 2.0 server that came installed with your operating system, chances are that it already includes `mod_ssl`. Use the package management tools bundled with your distribution to install `mod_ssl` if it is not present in your system.

When you build Apache 2.0 from source, you must pass the following options to enable and build `mod_ssl` at compile time.

```
--enable-ssl --with-ssl=/usr/local/ssl/install/openssl
```

This assumes that you installed OpenSSL in the location described in previous sections.

If you compiled `mod_ssl` statically into Apache, you can check whether it is present by issuing the following command, which provides a list of compiled-in modules:

```
# /usr/local/apache2/bin/httpd -l
```

The command assumes that you installed Apache in the `/usr/local/apache2` directory.

If `mod_ssl` was compiled as a dynamic loadable module, the following line must be added or uncommented to the configuration file:

```
LoadModule ssl_module modules/libmodssl.so
```

Managing Certificates

To have a working SSL server implementation, the first step is to create a server certificate. This section explains in detail how to create and manage certificates and keys by using the `openssl` command-line tool. For example, if you are using SSL for an

e-commerce site, encryption prevents customer data from eavesdroppers and the certificate enables customers to verify that you are who you claim to be.

The examples refer to the Unix version of the command-line program `openssl`. If you are running under Windows, you need to use `openssl.exe` instead and change the paths of the examples to use backslashes instead of forward slashes. The examples also assume that OpenSSL was installed in the path described earlier in the OpenSSL installation section.

Creating a Key Pair

You must have a public/private key pair before you can create a certificate request. Assume that the FQDN for the certificate you want to create is `www.example.com`. (You will need to substitute this name for the FQDN of the machine you have installed Apache on.) You can create the keys by issuing the following command:

```
# ./usr/local/ssl/install/bin/openssl genrsa -des3 -rand file1:file2:file3 \
     -out www.example.com.key 1024
```

`genrsa` indicates to OpenSSL that you want to generate a key pair.

`des3` indicates that the private key should be encrypted and protected by a pass phrase.

The `rand` switch is used to provide OpenSSL with random data to ensure that the generated keys are unique and unpredictable. Substitute *`file1`*, *`file2`*, and so on, for the path to several large, relatively random files for this purpose (such as a kernel image, compressed log files, and so on). This switch is not necessary on Windows because the random data is automatically generated by other means.

The `out` switch indicates where to store the results.

`1024` indicates the number of bits of the generated key.

The result of invoking this command looks like this:

```
625152  semi-random bytes loaded
Generating RSA private key, 1024 bit long modulus
.....++++++
.........................++++++
e is 65537 (0x10001)
Enter PEM pass phrase:
Verifying password - Enter PEM pass phrase:
```

As you can see, you will be asked to provide a pass phrase. Choose a secure one. The pass phrase is necessary to protect the private key and you will be asked for it whenever you want to start the server. You can choose not to protect the key. This is convenient because you will not need to enter the pass phrase during reboots, but it is highly insecure and a compromise of the server means a compromise of the key as well. In any case, you can choose to unprotect the key either by leaving out the `-des3` switch in the generation phase or by issuing the following command:

```
# ./usr/local/ssl/install/bin/openssl rsa -in www.example.com.key \
       -out www.example.com.key.unsecure
```

It is a good idea to back up the `www.example.com.key` file. You can learn about the contents of the key file by issuing the following command:

```
# ./usr/local/ssl/bin/openssl rsa -noout -text -in www.example.com.key
```


Creating a Certificate Signing Request

To get a certificate issued by a CA, you must submit what is called a *certificate signing request*. To create a request, issue the following command:

```
# ./usr/local/ssl/install/bin/openssl req -new -key www.example.com.key
 -out www.example.com.csr
```

You will be prompted for the certificate information:

```
Using configuration from /usr/local/ssl/install/openssl/openssl.cnf
Enter PEM pass phrase:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (eg, city) []: San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:.
Organizational Unit Name (eg, section) []:.
Common Name (eg, YOUR name) []:www.example.com
Email Address []:administrator@example.com
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

It is important that the `Common Name` field entry matches the address that visitors to your Web site will type in their browsers. This is one of the checks that the browser will

perform for the remote server certificate. If the names differ, a warning indicating the mismatch will be issued to the user.

The certificate is now stored in `www.example.com.csr`. You can learn about the contents of the certificate via the following command:

```
# ./usr/local/ssl/install/bin/openssl req -noout -text \
       -in www.example.com.csr
```

You can submit the certificate signing request file to a CA for processing. VeriSign and Thawte are two of those CAs. You can learn more about their particular submission procedures at their Web sites:

- **VeriSign:** `http://digitalid.verisign.com/server/apacheNotice.htm`
- **Thawte:** `http://www.thawte.com/certs/server/request.html`

Creating a Self-Signed Certificate

You can also create a self-signed certificate. That is, you can be both the issuer and the subject of the certificate. Although this is not very useful for a commercial Web site, it will enable you to test your installation of `mod_ssl` or to have a secure Web server while you wait for the official certificate from the CA.

```
# ./usr/local/ssl/install/bin/openssl x509 -req -days 30  \
-in www.example.com.csr -signkey www.example.com.key \
-out www.example.com.cert
```

You need to copy your certificate `www.example.com.cert` (either the one returned by the CA or your self-signed one) to `/usr/local/ssl/install/openssl/certs/` and your key to `/usr/local/ssl/install/openssl/private/`.

Protect your key file by issuing the following command:

```
# chmod 400 www.example.com.key
```

SSL Configuration

The previous sections introduced the (not-so-basic) concepts behind SSL and you have learned how to generate keys and certificates. Now, finally, you can configure Apache to support SSL. `mod_ssl` must either be compiled statically or, if you have compiled as a loadable module, the appropriate `LoadModule` directive must be present in the file.

If you compiled Apache yourself, a new Apache configuration file, named `ssl.conf`, should be present in the `conf/` directory. That file contains a sample Apache SSL configuration and is referenced from the main `httpd.conf` file via an `Include` directive.

If you want to start your configuration from scratch, you can add the following configuration snippet to your Apache configuration file:

```
Listen 80
Listen 443
<VirtualHost _default_:443>
ServerName www.example.com
SSLEngine on
SSLCertificateFile \
/usr/local/ssl/install/openssl/certs/www.example.com.cert
SSLCertificateKeyFile \
/usr/loca/ssl/install/openssl/certs/www.example.com.key
</VirtualHost>
```

With the previous configuration, you set up a new virtual host that will listen to port 443 (the default port for HTTPS) and you enable SSL on that virtual host with the `SSLEngine` directive.

You need to indicate where to find the server's certificate and the file containing the associated key. You do so by using `SSLCertificateFile` and `SSLCertificateKeyfile` directives.

Starting the Server

Now you can stop the server if it is running, and start it again. If your key is protected by a pass phrase, you will be prompted for it. After this, Apache will start and you should be able to connect securely to it via the `https://www.example.com/` URL.

If you compiled and installed Apache yourself, in many of the vendor configuration files, you can see that the SSL directives are surrounded by an `<IfDefine SSL>` block. That allows for conditional starting of the server in SSL mode. If you start the `httpd` server binary directly, you can pass it the `-DSSL` flag at startup. You can also use the `apachectl` script by issuing the `apachectl startssl` command. Finally, if you always want to start Apache with SSL support, you can just remove the `<ifDefine>` section and start Apache in the usual way.

If you are unable to successfully start your server, check the Apache error log for clues about what might have gone wrong. For example, if you cannot bind to the port, make sure that another Apache is not running already. You must have administrator privileges to bind to port 443; otherwise, you can change the port to 8443 and access the URL via `https://www.example.com:8443`.

Configuration Directives

`mod_ssl` provides comprehensive technical reference documentation. This information will not be reproduced here; rather, I will explain what is possible and which

configuration directives you need to use. You can then refer to the online SSL documentation bundled with Apache for the specific syntax or options.

Algorithms

You can control which ciphers and protocols are used via the `SSLCipherSuite` and `SSLProtocol` commands. For example, you can configure the server to use only strong encryption with the following configuration:

```
SSLProtocol all
SSLCipherSuite HIGH:MEDIUM
```

See the Apache documentation for a detailed description of all available ciphers and protocols.

Client Certificates

Similarly to how clients can verify the identity of servers using server certificates, servers can verify the identity of clients by requiring a client certificate and making sure that it is valid.

`SSLCACertificateFile` and `SSLCACertificatePath` are two Apache directives used to specify trusted Certificate Authorities. Only clients presenting certificates signed by these CAs will be allowed access to the server.

The `SSLCACertificateFile` directive takes a file containing a list of CAs as an argument. Alternatively, you could use the `SSLCACertificatePath` directive to specify a directory containing trusted CA files. Those files must have a specific format, described in the documentation. `SSLVerifyClient` enables or disables client certificate verification. `SSLVerifyDepth` controls the number of delegation levels allowed for a client certificate. The `SSLCARevocationFile` and `SSLCARevocationPath` directives enable you to specify certificate revocation lists to invalidate certificates.

Performance

SSLis a protocol that requires intensive calculations. `mod_ssl` and OpenSSL allow several ways to speed up the protocol by caching some of the information about the connection. You can cache certain settings using the `SSLSessionCache` and `SSLSessionCacheTimeout` directives. There is also built-in support for specialized cryptographic hardware that will perform the CPU-intensive computations and offload the main processor. The `SSLMutex` directive enables you to control the internal locking mechanism of the SSL engine. The `SSLRandomSeed` directive enables you to specify the mechanism to seed the random-number generator required for certain operations. The settings of both directives can have an impact on performance.

Logging

mod_ssl hooks into Apache's logging system and provides support for logging any SSL-related aspect of the request, ranging from the protocol used to the information contained in specific elements of a client certificate. This information can also be passed to CGI scripts via environment variables by using the StdEnvVars argument to the Options directive. SSLLog and SSLLogLevel enable you to specify where to store SSL-specific errors and which kind of errors to log. You can get a listing of the available SSL variables at http://httpd.apache.org/docs-2.0/ssl/ssl_compat.html.

The SSLOptions Directive

Many of these options can be applied in a per-directory or per-location basis. The SSL parameters might be renegotiated for those URLs. This can be controlled via the SSLOptions directive.

The SSLPassprase directive can be used to avoid having to enter a pass phrase at startup by designating an external program that will be invoked to provide it.

Access Control

The SSLRequireSSL directive enables you to force clients to access the server using SSL. The SSLRequire directive enables you to specify a set of rules that have to be met before the client is allowed access. SSLRequire syntax can be very complex, but itallows an incredible amount of flexibility. Listing 17.1 shows a sample configuration from the mod_ssl documentation that restricts access based on the client certificate and the network the request came from. Access will be granted if one of the following is met:

- The SSL connection does not use an export (weak) cipher or a NULL cipher, the certificate has been issued by a particular CA and for a particular group, and the access takes place during workdays (Monday to Friday) and working hours (8:00 a.m. to 8:00 p.m.).
- The client comes from an internal, trusted network.

You can check the documentation for SSLRequire for a complete syntax reference.

LISTING 17.1 SSLRequire Example

```
SSLRequire (    %{SSL_CIPHER} !~ m/^(EXP|NULL)-/ \
            and %{SSL_CLIENT_S_DN_O} eq "Snake Oil, Ltd." \
            and %{SSL_CLIENT_S_DN_OU} in {"Staff", "CA", "Dev"} \
            and %{TIME_WDAY} >= 1 and %{TIME_WDAY} <= 5 \
            and %{TIME_HOUR} >= 8 and %{TIME_HOUR} <= 20       ) \
           or %{REMOTE_ADDR} =~ m/^192\.76\.162\.[0-9]+$/
```

Reverse Proxy with SSL

Although at the time this book was written the SSL reverse proxy functionality was not included in `mod_ssl` for Apache 2.0, it is likely to be included in the future. That functionality enables you to encrypt the reverse proxy connection to backend servers and to perform client and server certificate authentication on that connection. The related directives are `SSLProxyMachineCertificatePath`, `SSLProxyMachineCertificateFile`, `SSLProxyVerify`, `SSLProxyVerifyDepth`, `SSLProxyCACertificatePath`, `SSLProxyEngine`, and `SSLProxyCACertificateFile`. Their syntax is similar to their regular counterparts. You can find more information about the Apache reverse proxy in Hour 15.

Problems with Specific Browser Versions

Some browsers have known problems with specific versions of the SSL protocol or certain features. Certain environment variables can be set to force specific behaviors. The following example, included in the default configuration file, is a workaround for bugs in the SSL implementation of Internet Explorer browsers.

```
SetEnvIf User-Agent ".*MSIE.*" nokeepalive ssl-unclean-shutdown \
downgrade-1.0 force-response-1.0
```

Summary

This hour explained the fundamentals of the SSL protocol and `mod_ssl`, the Apache module that implements support for SSL. You learned how to install and configure `mod_ssl` and the OpenSSL libraries, and how to use the `openssl` command-line tool for certificate and key generation and management. You can access the `mod_ssl` reference documentation for in-depth syntax explanation and additional configuration information. Bear in mind also that SSL is just part of maintaining a secure server, which includes applying security patches, OS configuration, access control, physical security, and so on.

Q&A

Q Can I have SSL with name-based virtual hosting?

A A question that comes up frequently is how to make name-based virtual hosts work with SSL. The answer is that you can't, at least currently. Name-based virtual hosts depend on the `Host` header of the HTTP request, but the certificate verification happens when the SSL connection is being established and no HTTP request can be sent. There is a protocol for upgrading an existing HTTP connection to TLS, but it is mostly unsupported by current browsers (see RFC 2817).

Q Can I use SSL with other protocols?

A `mod_ssl` implements the SSL protocol as a filter. Other protocols using the same Apache server can easily take advantage of the SSL.

Quiz

1. How can you prevent the prompting for a password at startup?
2. How can you use the `openssl` command-line tool to connect to an SSL-enabled server?

   The `openssl` command-line tool enables you to connect to SSL-enabled servers. Read the documentation and figure out how to do it. You can use the Unix man page for `openssl` or read the documentation at `http://www.openssl.org`.

Quiz Answers

1. You can use the `SSLPassPhrase` method to point to a program that will provide the pass phrase. The program should make the appropriate checks to make sure that it reveals the pass phrase only to Apache.

   Additionally, you could simply remove the password protection from the file containing the key, as described earlier in the hour. This has severe security implications, but it can be very convenient.

2.

   ```
   # openssl s_client -connect www.ibm.com:443
   ```

   You will see information related to the connection, certificates, ciphers, and so on. Then you can type

   ```
   GET / HTTP/1.0
   ```

   to get the contents of the index HTML page, similar to the way you learned in Hour 2, "Understanding Apache Internals," with `telnet`.

   You can configure many aspects of the connection, as explained in the documentation.

Related Directives

This section contains directives mentioned in this hour or that are related to topics discussed in this hour. You can consult the Apache reference documentation for comprehensive syntax information and usage.

Keys and Certificates

- **`SSLPassPhraseDialog`:** Alternative ways of specifying a pass phrase to decrypt key
- **`SSLCertificateFile`:** File containing server certificate
- **`SSLCertificateKeyFile`:** File containing server key
- **`SSLCertificateChainFile`:** File containing chain of certificates used to sign the server certificate
- **`SSLCACertificatePath`:** Path to a directory containing CA certificates for client authentication
- **`SSLCACertificateFile`:** Path to a file containing CA certificates for client authentication
- **`SSLCARevocationPath`:** Path to a directory containing CA for revoking client certificates
- **`SSLCARevocationFile`:** Path to a file containing CA for revoking client certificates
- **`SSLVerifyClient`:** Enable client certificate verification
- **`SSLVerifyDepth`:** Establish maximum depth to verify client certificates to

SSL Protocol

- **`SSLProtocol`:** Versions of SSL supported
- **`SSLCipherSuite`:** Ciphers supported
- **`SSLEngine`:** Enable SSL protocol engine
- **`SSLRequireSSL`:** Require client to connect to server using SSL
- **`SSLRequire`:** Require specific rules for client to connect

Performance

- **`SSLMutex`:** Locking mechanism
- **`SSLRandomSeed`:** Initialize random number generator
- **`SSLSessionCache`:** Specify an SSL-caching mechanism
- **`SSLSessionCacheTimeout`:** Caching sessions expiry time

Others

- **`SSLOptions`:** Control various aspects of SSL operation

Reverse Proxy

- **`SSLProxyMachineCertificatePath`**, **`SSLProxyMachineCertificateFile`**, **`SSLProxyVerify`**, **`SSLProxyVerifyDepth`**, **`SSLProxyCACertificatePath`**, **`SSLProxyCACertificateFile`:** Equivalent to their regular server counterparts, but related to the reverse proxy connection

Further Reading

An excellent, highly readable cryptography reference book is *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Second Edition, by Bruce Schneier; ISBN 0471117099.

A great book on the SSL protocol, and especially useful if you are programming with SSL libraries, is *SSL and TLS: Designing and Building Secure Systems*, by Eric Rescorla; ISBN 0201615983.

OpenSSL project: `http://www.openssl.org`

ModSSL project: `http://www.openssl.org`

OpenBSD, a free Unix server operating system with a focus on security: `http://www.openbsd.com`

Apache reference, by the original author of `mod_ssl`: `http://www.apacheref.com`

SSLv2 specification: `http://home.netscape.com/eng/security/SSL_2.html`

SSLv3 specification: `http://home.netscape.com/eng/ssl3/draft302.txt`

The following SSL-related RFCs can be obtained from `http://www.rfc-editor.org/`:

- Internet X.509 PKI: RFC 2459
- Transport Layer Security: RFC 2246
- Upgrading to TLS Within HTTP/1.1: RFC 2817