Cryptography for Secure Communications
Network Security with
OpenSSL
O'REILLY®
John Viega, Matt Messier & Pravir Chandra

Network Security
with OpenSSL

Network Security with OpenSSL

John Viega, Matt Messier, and Pravir Chandra

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Network Security with OpenSSL
by John Viega, Matt Messier, and Pravir Chandra

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Robert Denn
Production Editor: Colleen Gorman
Cover Designer: Ellie Volckhausen
Interior Designer: David Futato

Printing History:

June 2002: First Edition.

ISBN: 0-596-00270-X
[M] [8/02]

To the memory of Arthur J. Zoebelein,
former Chief of the Office of Cryptologic
Archives and History, National Security Agency

Jänicke, Ben Laurie, Richard Levitte, Bodo Möller, Ulf Möller, Andy Polyakov, Holger Reif, Paul Sutton, Geoff Thorpe, and Eric A. Young.

We also thank Sue Miller for encouraging us to write this book in the first place.

—John Viega, Matt Messier, and Pravir Chandra
March 2002
Fairfax, VA

CHAPTER 1
Introduction

In today's networked world, many applications need security, and cryptography is one of the primary tools for providing that security. The primary goals of cryptography, data confidentiality, data integrity, authentication, and non-repudiation (accountability) can be used to thwart numerous types of network-based attacks, including eavesdropping, IP spoofing, connection hijacking, and tampering. OpenSSL is a cryptographic library; it provides implementations of the industry's best-regarded algorithms, including encryption algorithms such as 3DES ("Triple DES"), AES and RSA, as well as message digest algorithms and message authentication codes.

Using cryptographic algorithms in a secure and reliable manner is much more difficult than most people believe. Algorithms are just building blocks in cryptographic protocols, and cryptographic protocols are notoriously difficult to get right. Cryptographers have a difficult time devising protocols that resist all known attacks, and the average developer tends to do a lot worse. For example, developers often try to secure network connections simply by encrypting data before sending it, then decrypting it on receipt. That strategy often fails to ensure the integrity of data. In many situations, attackers can tamper with data, and sometimes even recover it. Even when protocols are well designed, implementation errors are common. Most cryptographic protocols have limited applicability, such as secure online voting. However, protocols for securely communicating over an insecure medium have ubiquitous applicability. That's the basic purpose of the SSL protocol and its successor, TLS (when we generically refer to SSL, we are referring to both SSL and TLS): to provide the most common security services to arbitrary (TCP-based) network connections in such a way that the need for cryptographic expertise is minimized.

Ultimately, it would be nice if developers and administrators didn't need to know anything about cryptography or even security to protect their applications. It would be nice if security was as simple as linking in a different socket library when building a program. The OpenSSL library strives toward that ideal as much as possible, but in

reality, even the SSL protocol requires a good understanding of security principles to apply securely. Indeed, most applications using SSL are susceptible to attack.

Nonetheless, SSL certainly makes securing network connections much simpler. Using SSL doesn't require any understanding of how cryptographic algorithms work. Instead, you only need to understand the basic properties important algorithms have. Similarly, developers do not need to worry about cryptographic protocols; SSL doesn't require any understanding of its internal workings in order to be used. You only need to understand how to apply the algorithm properly.

The goal of this book is to document the OpenSSL library and how to use it properly. This is a book for practitioners, not for security experts. We'll explain what you need to know about cryptography in order to use it effectively, but we don't attempt to write a comprehensive introduction on the subject for those who are interested in why cryptography works. For that, we recommend *Applied Cryptography*, by Bruce Schneier (John Wiley & Sons). For those interested in a more technical introduction to cryptography, we recommend Menezes, van Oorschot, and Vanstone's *Handbook of Applied Cryptography* (CRC Press). Similarly, we do not attempt to document the SSL protocol itself, just its application. If you're interested in the protocol details, we recommend Eric Rescorla's *SSL and TLS* (Addison-Wesley).

Cryptography for the Rest of Us

For those who have never had to work with cryptography before, this section introduces you to the fundamental principles you'll need to know to understand the rest of the material in this book. First, we'll look at the problems that cryptography aims to solve, and then we'll look at the primitives that modern cryptography provides. Anyone who has previously been exposed to the basics of cryptography should feel free to skip ahead to the next section.

Goals of Cryptography

The primary goal of cryptography is to secure important data as it passes through a medium that may not be secure itself. Usually, that medium is a computer network.

There are many different cryptographic algorithms, each of which can provide one or more of the following services to applications:

Confidentiality (secrecy)
: Data is kept secret from those without the proper credentials, even if that data travels through an insecure medium. In practice, this means potential attackers might be able to see garbled data that is essentially "locked," but they should not be able to unlock that data without the proper information. In classic cryptography, the *encryption* (scrambling) algorithm was the secret. In modern cryptography, that isn't feasible. The algorithms are public, and cryptographic *keys* are used in the encryption and decryption processes. The only thing that needs to be secret is the key. In addition, as we will demonstrate a bit later, there are common cases in which not all keys need to be kept secret.

Integrity (anti-tampering)
: The basic idea behind data integrity is that there should be a way for the recipient of a piece of data to determine whether any modifications are made over a period of time. For example, integrity checks can be used to make sure that data sent over a wire isn't modified in transit. Plenty of well-known checksums exist that can detect and even correct simple errors. However, such checksums are poor at detecting skilled intentional modifications of the data. Several cryptographic checksums do not have these drawbacks if used properly. Note that encryption does not ensure data integrity. Entire classes of encryption algorithms are subject to "bit-flipping" attacks. That is, an attacker can change the actual value of a bit of data by changing the corresponding encrypted bit of data.

Authentication
: Cryptography can help establish identity for authentication purposes.

Non-repudiation
: Cryptography can enable Bob to prove that a message he received from Alice actually came from Alice. Alice can essentially be held accountable when she sends Bob such a message, as she cannot deny (repudiate) that she sent it. In the real world, you have to assume that an attacker does not compromise particular cryptographic keys. The SSL protocol does not support non-repudiation, but it is easily added by using digital signatures.

These simple services can be used to stop a wide variety of network attacks, including:

Snooping (passive eavesdropping)
: An attacker watches network traffic as it passes and records interesting data, such as credit card information.

Tampering
: An attacker monitors network traffic and maliciously changes data in transit (for example, an attacker may modify the contents of an email message).

Spoofing
: An attacker forges network data, appearing to come from a different network address than he actually comes from. This sort of attack can be used to thwart systems that authenticate based on host information (e.g., an IP address).

Hijacking
: Once a legitimate user authenticates, a spoofing attack can be used to "hijack" the connection.

Capture-replay
: In some circumstances, an attacker can record and replay network transactions to ill effect. For example, say that you sell a single share of stock while the price is high. If the network protocol is not properly designed and secured, an attacker

could record that transaction, then replay it later when the stock price has dropped, and do so repeatedly until all your stock is gone.

Many people assume that some (or all) of the above attacks aren't actually feasible in practice. However, that's far from the truth. Especially due to tool sets such as dsniff (*http://www.monkey.org/~dugsong/dsniff/*), it doesn't even take much experience to launch all of the above attacks if access to any node on a network between the two endpoints is available. Attacks are equally easy if you're on the same local network as one of the endpoints. Talented high school students who can use other people's software to break into machines and manipulate them can easily manage to use these tools to attack real systems.

Traditionally, network protocols such as HTTP, SMTP, FTP, NNTP, and Telnet don't provide adequate defenses to the above attacks. Before electronic commerce started taking off in mid-1990, security wasn't really a large concern, especially considering the Internet's origins as a platform for sharing academic research and resources. While many protocols provided some sort of authentication in the way of password-based logins, most of them did not address confidentiality or integrity at all. As a result, all of the above attacks were possible. Moreover, authentication information could usually be among the information "snooped" off a network.

SSL is a great boon to the traditional network protocols, because it makes it easy to add transparent confidentiality and integrity services to an otherwise insecure TCP-based protocol. It can also provide authentication services, the most important being that clients can determine if they are talking to the intended server, not some attacker that is spoofing the server.

Cryptographic Algorithms

The SSL protocol covers many cryptographic needs. Sometimes, though, it isn't good enough. For example, you may wish to encrypt HTTP cookies that will be placed on an end user's browser. SSL won't help protect the cookies while they're being stored on that disk. For situations like this, OpenSSL exports the underlying cryptographic algorithms used in its implementation of the SSL protocol.

Generally, you should avoid using cryptographic algorithms directly if possible. You're not likely to get a totally secure system simply by picking an algorithm and applying it. Usually, cryptographic algorithms are incorporated into cryptographic protocols. Plenty of nonobvious things can be wrong with a protocol based on cryptographic algorithms. That is why it's better to try to find a well-known cryptographic protocol to do what you want to do, instead of inventing something yourself. In fact, even the protocols invented by cryptographers often have subtle holes.

If not for public review, most protocols in use would be insecure. Consider the original WEP protocol for IEEE 802.11 wireless networking. WEP (Wired Equivalent Privacy) is the protocol that is supposed to provide the same level of security for data that physical lines provide. It is a challenge, because data is transmitted through the

air, instead of across a wire. WEP was designed by veteran programmers, yet without soliciting the opinions of any professional cryptographers or security protocol developers. Although to a seasoned developer with moderate security knowledge the protocol looked fine, in reality, it was totally lacking in security.

Nonetheless, sometimes you might find a protocol that does what you need, but can't find an implementation that suits your needs. Alternatively, you might find that you do need to come up with your own protocol. For those cases, we do document the SSL cryptographic API.

Five types of cryptographic algorithms are discussed in this book: symmetric key encryption, public key encryption, cryptographic hash functions, message authentication codes, and digital signatures.

Symmetric key encryption

Symmetric key algorithms encrypt and decrypt data using a single key. As shown in Figure 1-1, the key and the plaintext message are passed to the encryption algorithm, producing ciphertext. The result can be sent across an insecure medium, allowing only a recipient who has the original key to decrypt the message, which is done by passing the ciphertext and the key to a decryption algorithm. Obviously, the key must remain secret for this scheme to be effective.

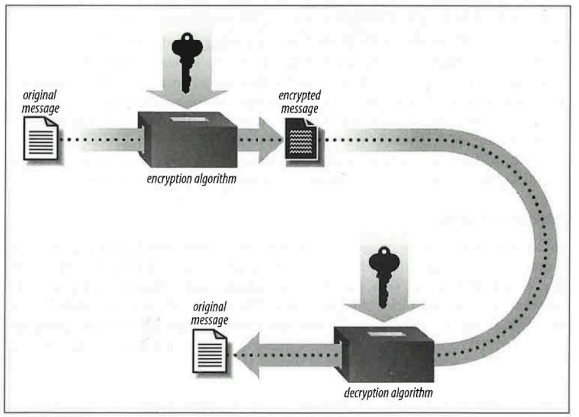

Figure 1-1. Symmetric key cryptography

The primary disadvantage of symmetric key algorithms is that the key must remain secret at all times. In particular, exchanging secret keys can be difficult, since you'll usually want to exchange keys on the same medium that you're trying to use encryption to protect. Sending the key in the clear before you use it leaves open the possibility of an attacker recording the key before you even begin to send data.

One solution to the key distribution problem is to use a cryptographic key exchange protocol. OpenSSL provides the Diffie-Hellman protocol for this purpose, which allows for key agreement without actually divulging the key on the network. However, Diffie-Hellman does not guarantee the identity of the party with whom you are exchanging keys. Some sort of authentication mechanism is necessary to ensure that you don't accidentally exchange keys with an attacker.

Right now, Triple DES (usually written 3DES, or sometimes DES3) is the most conservative symmetric cipher available. It is in wide use, but AES, the new Advanced Encryption Standard, will eventually replace it as the most widely used cipher. AES is certainly faster than 3DES, but 3DES has been around a lot longer, and thus is a more conservative choice for the ultra-paranoid. It is worth mentioning that RC4 is widely supported by existing clients and servers. It is faster than 3DES, but is difficult to set up properly (don't worry, SSL uses RC4 properly). For purposes of compatibility with existing software in which neither AES nor 3DES are supported, RC4 is of particular interest. We don't recommend supporting other algorithms without a good reason. For the interested, we discuss cipher selection in Chapter 6.

Security is related to the length of the key. Longer key lengths are, of course, better. To ensure security, you should only use key lengths of 80 bits or higher. While 64-bit keys may be secure, they likely will not be for long, whereas 80-bit keys should be secure for at least a few years to come. AES supports only 128-bit keys and higher, while 3DES has a fixed 112 bits of effective security.* Both of these should be secure for all cryptographic needs for the foreseeable future. Larger keys are probably unnecessary. Key lengths of 56 bits (regular DES) or less (40-bit keys are common) are too weak; they have proven to be breakable with a modest amount of time and effort.

Public key encryption

Public key cryptography suggests a solution to the key distribution problem that plagues symmetric cryptography. In the most popular form of public key cryptography, each party has two keys, one that must remain secret (the *private key*) and one that can be freely distributed (the *public key*). The two keys have a special mathematical relationship. For Alice to send a message to Bob using public key encryption (see Figure 1-2), Alice must first have Bob's public key. She then encrypts her message using Bob's public key, and delivers it. Once encrypted, only someone who has Bob's private key can successfully decrypt the message (hopefully, that's only Bob).

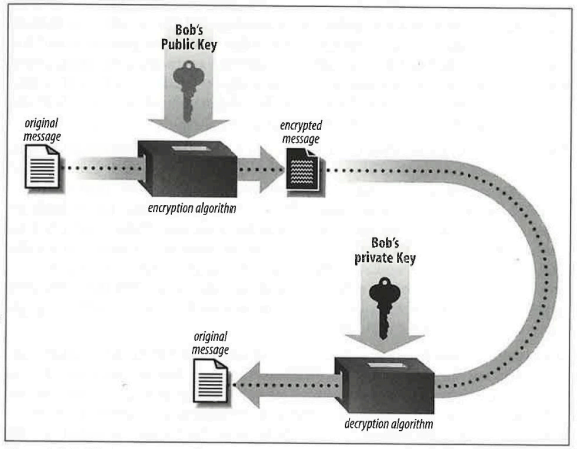

Figure 1-2. Public key cryptography

Public key encryption solves the problem of key distribution, assuming there is some way to find Bob's public key and ensure that the key really does belong to Bob. In practice, public keys are passed around with a bunch of supporting information called a *certificate*, and those certificates are validated by trusted third parties. Often, a trusted third party is an organization that does research (such as credit checks) on people who wish to have their certificates validated. SSL uses trusted third parties to help address the key distribution problem.

Public key cryptography has a significant drawback, though: it is intolerably slow for large messages. Symmetric key cryptography can usually be done quickly enough to encrypt and decrypt all the network traffic a machine can manage. Public key cryptography is generally limited by the speed of the cryptography, not the bandwidth going into the computer, particularly on server machines that need to handle multiple connections simultaneously.

As a result, most systems that use public key cryptography, SSL included, use it as little as possible. Generally, public key encryption is used to agree on an encryption key for a symmetric algorithm, and then all further encryption is done using the

* 3DES provides 168 bits of security against brute-force attacks, but there is an attack that reduces the effective security to 112 bits. The enormous space requirements for that attack makes it about as practical as brute force (which is completely impractical in and of itself).

symmetric algorithm. Therefore, public key encryption algorithms are primarily used in key exchange protocols and when non-repudiation is required.

RSA is the most popular public key encryption algorithm. The Diffie-Hellman key exchange protocol is based on public key technology and can be used to achieve the same ends by exchanging a symmetric key, which is used to perform actual data encryption and decryption. For public key schemes to be effective, there usually needs to be an authentication mechanism involving a trusted third party that is separate from the encryption itself. Most often, digital signature schemes, which we discuss below, provide the necessary authentication.

Keys in public key algorithms are essentially large numbers with particular properties. Therefore, bit length of keys in public key ciphers aren't directly comparable to symmetric algorithms. With public key encryption algorithms, you should use keys of 1,024 bits or more to ensure reasonable security. 512-bit keys are probably too weak. Anything larger than 2,048 bits may be too slow, and chances are it will not buy security that is much more practical. Recently, there's been some concern that 1,024-bit keys are too weak, but as of this writing, there hasn't been conclusive proof. Certainly, 1,024 bits is a bare minimum for practical security from short-term attacks. If your keys potentially need to stay protected for years, then you might want to go ahead and use 2,048-bit keys.

When selecting key lengths for public key algorithms, you'll usually need to select symmetric key lengths as well. Recommendations vary, but we recommend using 1,024-bit keys when you are willing to work with symmetric keys that are less than 100 bits in length. If you're using 3DES or 128-bit keys, we recommend 2,048-bit public keys. If you are paranoid enough to be using 192-bit keys or higher, we recommend using 4,096-bit public keys.

Requirements for key lengths change if you're using elliptic curve cryptography (ECC), which is a modification of public key cryptography that can provide the same amount of security using faster operations and smaller keys. OpenSSL currently doesn't support ECC, and there may be some lingering patent issues for those who wish to use it. For developers interested in this topic, we recommend the book *Implementing Elliptic Curve Cryptography*, by Michael Rosing (Manning).

Cryptographic hash functions and Message Authentication Codes

Cryptographic hash functions are essentially checksum algorithms with special properties. You pass data to the hash function, and it outputs a fixed-size checksum, often called a *message digest*, or simply digest for short. Passing identical data into the hash function twice will always yield identical results. However, the result gives away no information about the data input to the function. Additionally, it should be practically impossible to find two inputs that produce the same message digest. Generally, when we discuss such functions, we are talking about one-way functions. That is, it should not be possible to take the output and algorithmically reconstruct

the input under any circumstances. There are certainly reversible hash functions, but we do not consider such things in the scope of this book.

For general-purpose usage, a minimally secure cryptographic hash algorithm should have a digest twice as large as a minimally secure symmetric key algorithm. MD5 and SHA1 are the most popular one-way cryptographic hash functions. MD5's digest length is only 128 bits, whereas SHA1's is 160 bits. For some uses, MD5's key length is suitable, and for others, it is risky. To be safe, we recommend using only cryptographic hash algorithms that yield 160-bit digests or larger, unless you need to support legacy algorithms. In addition, MD5 is widely considered "nearly broken" due to some cryptographic weaknesses in part of the algorithm. Therefore, we recommend that you avoid using MD5 in any new applications.

Cryptographic hash functions have been put to many uses. They are frequently used as part of a password storage solution. In such circumstances, logins are checked by running the hash function over the password and some additional data, and checking it against a stored value. That way, the server doesn't have to store the actual password, so a well-chosen password will be safe even if an attacker manages to get a hold of the password database.

Another thing people like to do with cryptographic hashes is to release them alongside a software release. For example, OpenSSL might be released alongside a MD5 checksum of the archive. When you download the archive, you can also download the checksum. Then you can compute the checksum over the archive and see if the computed checksum matches the downloaded checksum. You might hope that if the two checksums match, then you securely downloaded the actual released file, and did not get some modified version with a Trojan horse in it. Unfortunately, that isn't the case, because there is no secret involved. An attacker can replace the archive with a modified version, and replace the checksum with a valid value. This is possible because the message digest algorithm is public, and there is no secret information input to it.

If you share a secret key with the software distributor, then the distributor could combine the archive with the secret key to produce a message digest that an attacker shouldn't be able to forge, since he wouldn't have the secret. Schemes for using *keyed hashes*, i.e., hashes involving a secret key, are called *Message Authentication Codes* (MACs). MACs are often used to provide message integrity for general-purpose data transfer, whether encrypted or not. Indeed, SSL uses MACs for this purpose.

The most widely used MAC, and the only one currently supported in SSL and in OpenSSL, is HMAC. HMAC can be used with any message digest algorithm.

Digital signatures

For many applications, MACs are not very useful, because they require agreeing on a shared secret. It would be nice to be able to authenticate messages without needing to share a secret. Public key cryptography makes this possible. If Alice signs a message with her secret signing key, then anyone can use her public key to verify that she

signed the message. RSA provides for digital signing. Essentially, the public key and private key are interchangeable. If Alice encrypts a message with her private key, anyone can decrypt it. If Alice didn't encrypt the message, using her public key to decrypt the message would result in garbage.

There is also a popular scheme called DSA (the Digital Signature Algorithm), which the SSL protocol and the OpenSSL library both support.

Much like public key encryption, digital signatures are very slow. To speed things up, the algorithm generally doesn't operate on the entire message to be signed. Instead, the message is cryptographically hashed, and then the hash of the message is signed. Nonetheless, signature schemes are still expensive. For this reason, MACs are preferable if any sort of secure key exchange has taken place.

One place where digital signatures are widely used is in certificate management. If Alice is willing to validate Bob's certificate, she can sign it with her public key. Once she's done that, Bob can attach her signature to his certificate. Now, let's say he gives the certificate to Charlie, and Charlie does not know that Bob actually gave him the certificate, but he would believe Alice if she told him the certificate belonged to Bob. In this case, Charlie can validate Alice's signature, thereby demonstrating that the certificate does indeed belong to Bob.

Since digital signatures are a form of public key cryptography, you should be sure to use key lengths of 1,024 bits or higher to ensure security.

Overview of SSL

SSL is currently the most widely deployed security protocol. It is the security protocol behind secure HTTP (HTTPS), and thus is responsible for the little lock in the corner of your web browser. SSL is capable of securing any protocol that works over TCP.

An SSL transaction (see Figure 1-3) starts with the client sending a handshake to the server. In the server's response, it sends its certificate. As previously mentioned, a certificate is a piece of data that includes a public key associated with the server and other interesting information, such as the owner of the certificate, its expiration date, and the fully qualified domain name* associated with the server.

During the connection process, the server will prove its identity by using its private key to successfully decrypt a challenge that the client encrypts with the server's public key. The client needs to receive the correct unencrypted data to proceed. Therefore, the server's certificate can remain public—an attacker would need a copy of the certificate as well as the associated private key in order to masquerade as a known server.

* By fully qualified, we mean that the server's hostname is written out in a full, unambiguous manner that includes specifying the top-level domain. For example, if our web server is named "www", and our corporate domain is "securesw.com", then the fully qualified domain name for that host is "www.securesw.com". No abbreviation of this name would be considered fully qualified.

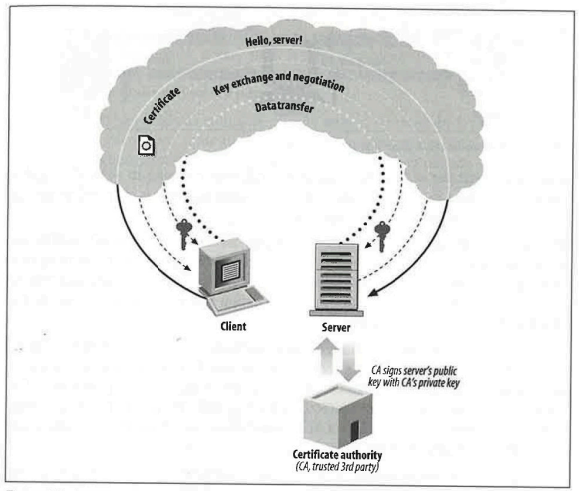

Figure 1-3. An overview of direct communication in SSL

However, an attacker could always intercept server messages and present the attacker's certificate. The data fields of the forged certificate can look legitimate (such as the domain name associated with the server and the name of the entity associated with the certificate). In such a case, the attacker might establish a proxy connection to the intended server, and then just eavesdrop on all data. Such an attack is called a "man-in-the-middle" attack and is shown in Figure 1-4. To thwart a man-in-the-middle attack completely, the client must not only perform thorough validation of the server certificate, but also have some way of determining whether the certificate itself is trustworthy. One way to determine trustworthiness is to hardcode a list of valid certificates into the client. The problem with this solution is that it is not scalable. Imagine needing the certificate for every secure HTTP server you might wish to use on the net stored in your web browser before you even begin surfing.

The practical solution to this problem is to involve a trusted third party that is responsible for keeping a database of valid certificates. A trusted third party, called a *Certification Authority*, signs valid server certificates using its private key. The signature indicates that the Certification Authority has done a background check on the

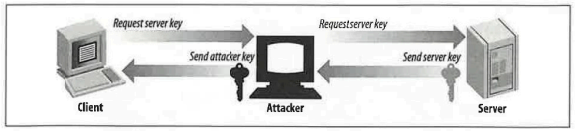

Figure 1-4. A man-in-the-middle attack

entity that owns the certificate being presented, thus ensuring to some degree that the data presented in the certificate is accurate. That signature is included in the certificate, and is presented at connection time.

The client can validate the authority's signature, assuming that it has the public key of the Certification Authority locally. If that check succeeds, the client can be reasonably confident the certificate is owned by an entity known to the trusted third party, and can then check the validity of other information stored in the certificate, such as whether the certificate has expired.

Although rare, the server can also request a certificate from the client. Before certificate validation is done, client and server agree on which cryptographic algorithms to use. After the certificate validation, client and server agree upon a symmetric key using a secure key agreement protocol (data is transferred using a symmetric key encryption algorithm). Once all of the negotiations are complete, the client and server can exchange data at will.

The details of the SSL protocol get slightly more complex. Message Authentication Codes are used extensively to ensure data integrity. Additionally, during certificate validation, a party can go to the Certification Authority for *Certificate Revocation Lists* (CRLs) to ensure that certificates that appear valid haven't actually been stolen.

We won't get into the details of the SSL protocol (or its successor, TLS). For our purposes, we can treat everything else as a black box. Again, if you are interested in the details, we recommend Eric Rescorla's book *SSL and TLS*.

Problems with SSL

SSL is an excellent protocol. Like many tools, it is effective in the hands of someone who knows how to use it well, but is easy to misuse. There are many pitfalls that people fall into when deploying SSL, most of which can be avoided with a bit of work.

Efficiency

SSL is a lot slower than a traditional unsecured TCP/IP connection. This problem is a direct result of providing adequate security. When a new SSL session is being established, the server and the client exchange a sizable amount of information that

is required for them to authenticate each other and agree on a key to be used for the session. This initial *handshake* involves heavy use of public key cryptography, which, as we've already mentioned, is very slow. It's also the biggest slowdown when using SSL. On current high-end PC hardware, OpenSSL struggles to make 100 connections per second under real workloads.

Once the initial handshake is complete and the session is established, the overhead is significantly reduced, but some of it still remains in comparison with an unsecured TCP/IP connection. Specifically, more data is transferred than normal. Data is transmitted in packets, which contain information required by the SSL protocol as well as any padding required by the symmetric cipher that is in use. Of course, there is the overhead of encrypting and decrypting the data as well, but the good news is that a symmetric cipher is in use, so it usually isn't a bottleneck. The efficiency of symmetric cryptography can vary greatly based on the algorithms used and the strength of the keys. However, even the slowest algorithms are efficient enough that they are rarely a bottleneck at all.

Because of the inefficiency of public key cryptography, many people decide not to use SSL when they realize it can't handle a large enough load. Some people go without security at all, which is obviously not a good idea. Other people try to design their own protocols to compensate. This is a bad idea, because there are many nonobvious pitfalls that can besiege you. Protocols that aren't designed by a skilled cryptographer inevitably have problems. SSL's design does consider efficiency; it simply isn't willing to sacrifice security for a speed improvement. You should be skeptical of using protocols that are more efficient.

There are ways to ameliorate this problem without abandoning the protocol. SSL does support a connection resumption mechanism so that clients that reconnect shortly after disconnecting can do so without incurring the full overhead of establishing a connection. While that is useful for HTTP,* it often isn't effective for other protocols.

Cryptographic acceleration hardware

One common approach for speeding up SSL is to use hardware acceleration. Many vendors provide PCI cards that can unload the burden of cryptographic operations from your processor, and OpenSSL supports most of them. We discuss the specifics of using hardware acceleration in Chapter 4.

* As is HTTP keepalive, which is a protocol option to keep sockets open for a period of time after a request is completed, so that the connection may be reused if another request to the same server follows in short order.

Load balancing

Another popular option for managing efficiency concerns with SSL is *load balancing*, which is simply distributing connections transparently across multiple machines, such that the group of machines appears as a single machine to the outside world for all intents and purposes. This can be a more cost-effective solution than accelerator cards, especially if you already have hardware lying around. Often, however, load balancing requires more work to ensure that persistent data is readily available to all servers on the backend. Another problem with load balancing is that many solutions route new connections to arbitrary machines, which can remove most of the benefit of connection resumption, since few clients will actually connect to the original machine during reconnection.

One simple load balancing mechanism is round-robin DNS, in which multiple IP addresses are assigned to a single DNS name. In response to DNS lookups, the DNS server cycles through all the addresses for that DNS name before giving out the same address twice. This is a popular solution because it is low-cost, requiring no special hardware. Connection resumption generally works well with this solution, since machines tend to keep a short-term memory of DNS results.

One problem with this solution is that the DNS server handles the load management, and takes no account of the actual load on individual servers. Additionally, large ISPs can perform DNS caching, causing an uneven distribution of load. To solve that problem, entries must be set to expire frequently, which increases the load on the DNS server.

Hardware load balancers vary in price and features. Those that can remember outside machines and map them to the same internal machine across multiple connections tend to be more expensive, but also more effective for SSL.

Version 0.9.7 of OpenSSL adds new functionality that allows applications to handle load balancing by way of manipulating session IDs. Sessions are a subset of operating parameters for an SSL connection, which we'll discuss in more detail in Chapter 5.

Keys in the Clear

In a typical SSL installation, the server maintains credentials so that clients can authenticate the server. In addition to a certificate that is presented at connection time, the server also maintains a private key, which is necessary for establishing that the server presenting a certificate is actually presenting its own certificate.

That private key needs to live somewhere on the server. The most secure solution is to use cryptographic acceleration hardware. Most of these devices can generate and store key material, and additionally prevent the private key from being accessed by an attacker who has broken into the machine. To do this, the private key is used only on the card, and is not allowed off except under special circumstances.

In cases in which hardware solutions aren't feasible, there is no absolute way to protect the private key from an attacker who has obtained root access, because, at the very least, the key must be unencrypted in memory when handling a new connection.* If an attacker has root, she can generally attach a debugger to the server process, and pull out the unencrypted key.

There are two options in these situations. First, you can simply keep the key unencrypted on disk. This is the easiest solution, but it also makes the job of an attacker simple if he has physical access, since he can power off the machine and pull out the disk, or simply reboot to single-user mode. Alternatively, you can keep the key encrypted on disk using a passphrase, which an administrator must type when the SSL server starts. In such a situation, the key will only be unencrypted in the address space of the server process, and thus won't be available to someone who can shut the machine off and directly access the disk.

Furthermore, many attackers are looking for low-hanging fruit, and will not likely go after the key even if they have the skills to do so. The downside to this solution is that unattended reboots are not possible, because whenever the machine restarts (or the SSL server process crashes), someone must type in the passphrase, which is often not very practical, especially in a lights-out environment. Storing the key in the clear obviously does not exhibit this problem.

In either case, your best defense is to secure the host and your network with the best available lockdown techniques (including physical lockdown techniques). Such solutions are outside the scope of this book.

What exactly does it mean if the server's private key is compromised? The most obvious result is that the attacker can masquerade as the server, which we discuss in the next section. Another result (which may not be as obvious) is that all past communications that used the key can likely be decrypted. If an attacker is able to compromise a private key, it is also likely that the attacker could have recorded previous communications. The solution to this problem is to use *ephemeral keying*. This means a temporary key pair is generated when a new SSL session is created. This is then used for key exchange and is subsequently destroyed. By using ephemeral keying, it is possible to achieve *forward secrecy*, meaning that if a key is compromised, messages encrypted with previous keys will not be subject to attack.† We discuss ephemeral keying and forward secrecy in more detail in Chapter 5.

* Some operating systems (particularly "trusted" OSs) can provide protection in such cases, assuming no security problems are in the OS implementation. Linux, Windows, and most of the BSD variants offer no such assurance.

† Note that if you are implementing a server in particular, it is often not possible to get perfect forward secrecy with SSL, since many clients don't support Diffie-Hellman, and because using cryptographically strong ephemeral RSA keys violates the protocol specification.

Bad Server Credentials

A server's private key can be stolen. In such a case, an attacker can usually masquerade as the server with impunity. Additionally, Certification Authorities sometimes sign certificates for people who are fraudulently representing themselves, despite the efforts made by the CA to validate all of the important information about the party that requests the certificate signing.* For example, in early 2001, VeriSign signed certificates that purported to belong to Microsoft, when in reality they did not. However, since they had been signed by a well-known Certification Authority, they would look authentic to anyone validating the signature on those certificates.

SSL has a mechanism for thwarting these problems: Certificate Revocation Lists. Once the Certification Authority learns that a certificate has been stolen or signed inappropriately, the Authority adds the certificate's serial number to a CRL. The client can access CRLs and validate them using the CA's certificate, since the server signs CRLs with its private key.

One problem with CRLs is that windows of vulnerability can be large. It can take time for an organization to realize that a private key may have been stolen and to notify the CA. Even when that happens, the CA must update its CRLs, which generally does not happen in real time (the time it takes depends on the CA). Then, once the CRLs are updated, the client must download them in order to detect that a presented certificate has been revoked. In most situations, clients never download or update CRLs. In such cases, compromised certificates tend to remain compromised until they expire.

There are several reasons for this phenomenon. First, CRLs tend to be large enough that they can take significant time to download, and can require considerable storage space locally, especially when the SSL client is an embedded device with limited storage capacity. The Online Certificate Status Protocol (OCSP), specified in RFC 2560, addresses these problems. Unfortunately, this is not yet a widely accepted standard protocol, nor is it likely to become so anytime soon. Additionally, the only version that is widely deployed has serious security issues (see Chapter 3 for more information). OpenSSL has only added OCSP support in Version 0.9.7, and few CAs even offer it as a service. Other authorities have facilities for incremental updates to CRLs, allowing for minimal download times, but that solution still requires space on the client, or some sort of caching server.

These solutions all require the CA's server to be highly available if clients are to have up-to-the-minute information. Some clients will be deployed in environments where

* Actually, a Registration Authority (RA) is responsible for authenticating information about the CA's customers. The CA can be its own RA, or it can use one or more third-party RAs. From the perspective of the consumer of certificates, the RA isn't really an important concept, so we will just talk about CAs to avoid confusion, even though it is technically not accurate.

a constant link to the CA is not possible. In addition, the need to query the CA can add significant latency to connection times that can be intolerable to the end user.

Another problem is that there is no standard delivery mechanism specified for CRLs. As a result, OpenSSL in particular does not provide a simple way to access CRL information, not even from VeriSign, currently the most popular CA. One common method of CRL (and certificate) distribution is using the Lightweight Directory Access Protocol (LDAP). LDAP provides a hierarchical structure for storing such information and fits nicely for PKI distribution.

Due to the many problems surrounding CRLs, it becomes even more important to take whatever measures are feasible to ensure that SSL private keys are not stolen. At the very least, you should put intrusion detection systems in place to detect compromises of your private key so that you can report the compromise to the CA quickly.

Certificate Validation

CRLs aren't useful if a client isn't performing adequate validation of server certificates to begin with. Often, they don't. Certainly, for SSL to work at all, the client must be able to extract the public key from a presented certificate, and the server must have a private key that corresponds with that public key. However, there is no mechanism to force further validation. As a result, man-in-the-middle attacks are often feasible.

First, developers must decide which Certification Authorities should be trusted, and must locate the certificates associated with each of those authorities. That's more effort than most developers are willing to exert. As a result, many applications using SSL are at the mercy of man-in-the-middle attacks.

Second, even those applications that install CA certificates and use them to validate server certificates often fail to perform adequate checking on the contents of the certificate. As a result, such systems are susceptible to man-in-the-middle attacks in which the attacker gets his hands on credentials that will look legitimate to the client, such as a stolen set of credentials in which the certificate is signed by the CA that has not yet appeared on any CRLs.

The best solution for thwarting this problem depends on the authentication needs of the client. Many applications can expect that they will only legitimately talk to a small set of servers. In such a case, you can check appropriate fields in the certificate against a white list of valid server names. For example, you might allow any certificate signed by VeriSign in which the fully qualified domain name mentioned in the certificate ends with "yourcompany.com". Another option is to hardcode a list of known server certificates. However, this is a far more difficult solution to manage if you ever wish to add servers.

Additionally, if you do not wish to trust the authentication mechanisms of the established CAs, you could consider running your own CA, which we discuss in

Chapter 3 (of course, we are assuming you control both the client and server code in such a situation). In environments where you expect that anyone can set up their own server, and thus managing DNS space or your own Certification Authority is not feasible, then the best you can do is ensure that the DNS address for the server that the client tried to contact is the same as the one presented in the certificate. If that is true, and the certificate was signed by a valid CA, everything should be fine if the certificate was not stolen or fraudulently obtained.

Poor Entropy

In the SSL protocol, both the client and the server need to generate random data for keys and other secrets. The data must be generated in such a way that a knowledgeable attacker cannot guess anything about it. SSL implementations usually generate such data using a *pseudorandom number generator* (PRNG). PRNGs are deterministic algorithms that produce a series of random-looking numbers. Classical PRNGs are not suitable for use in security-critical situations. Instead, SSL implementations use "cryptographic" PRNGs, which work in security-critical situations, as long as they are "seeded" properly.

A *seed* is a piece of data fed to the PRNG to get it going. Given a single, known seed at startup, the PRNG should produce a predictable set of outputs. That is, if you seed the PRNG and ask for three random numbers, reseed with the same value, and then ask for three more random numbers, the first three numbers and the second three numbers should be identical.

The seed itself must be a random number, but it can't just be a cryptographically random number. It must be truly unguessable to keep the PRNG outputs unguessable. *Entropy* is a measurement of how much unguessable information actually exists in data from the point of view of an attacker who might be able to make reasonable guesses about the state of the machine on which the number is stored. If a single bit is just as likely to be a 0 as a 1, then it is one bit of entropy. If you have 128 bits of data, it can have up to 128 bits of entropy. However, it may have as little as 0 bits of entropy—as would be the case if the data's value is public knowledge. The work an attacker must do to guess a piece of data is directly related to how much entropy there is in the data. If the data has 4 bits of entropy, then the attacker has a 1 in 2^4 chance (1 in 16) chance of guessing right the first time. Additionally, within 16 guesses, the attacker will have tried the right value (On average, he will find the right value in 8 guesses). If the data has 128 bits of entropy in it, then the attacker will need, on average 2^{127} guesses to find the seed, which is such a large number as to be infeasible for all practical purposes. In practice, if you're using 128-bit keys, it's desirable to use a seed with 128 bits of entropy or more. Anything less than 64 bits of entropy can probably be broken quickly by an organization with a modest hardware budget.

To illustrate, in 1996, Ian Goldberg and David Wagner found a problem with the way Netscape was seeding its PRNG in its implementation of SSLv2. Netscape was

using three inputs hashed with the MD5 message digest algorithm, the time of day, the process ID, and the parent process ID. None of these values is particularly random. At most, their PRNG seed could have had 47 bits of entropy. A clever attacker could decrease that substantially. Indeed, in practice, Goldberg and Wagner were able to compromise real SSL sessions within 25 seconds.

If you try to use OpenSSL without bothering to seed the random number generator, the library will complain. However, the library has no real way to know whether the seed you give it contains enough entropy. Therefore, you must have some idea how to get entropy. There are certainly hardware devices that do a good job of collecting it, including most of the cryptographic accelerator cards. However, in many cases hardware is impractical, because your software will be deployed across a large number of clients, most of whom will have no access to such devices.

Many software tricks are commonly employed for collecting entropy on a machine. They tend to work by indirectly measuring random information in external events that affect the machine. You should never need to worry about those actual techniques. Instead, use a package that harvests entropy for you. Many Unix-based operating systems now come with a random device, which provides entropy harvested by the operating system. On other Unix systems, you can use tools such as EGADS (*http://www.securesw.com/egads/*), which is a portable entropy collection system.* EGADS also works on Windows systems.

If you're interested in the entropy harvesting techniques behind random devices and tools like EGADS, see Chapter 10 of the book *Building Secure Software* by John Viega and Gary McGraw (Addison-Wesley).

Insecure Cryptography

While Version 3 of the SSL protocol and TLS are believed to be reasonably secure if used properly,† SSLv2 (Version 2) had fundamental design problems that led to wide-ranging changes in subsequent versions (Version 1 was never publicly deployed). For this reason, you should not support Version 2 of the protocol, just to ensure that an attacker does not launch a network attack that causes the client and server to settle upon the insecure version of the protocol. All you need to do is intercept the connection request and send a response that makes it look like a v3 server does not exist. The client will then try to connect using Version 2 of the protocol.

* We realize that Linux isn't technically a Unix operating system, since it is not derived from the original Unix code base. However, we feel the common usage of the term Unix extends to any Unix-like operating system, and that's how we use this term throughout the book.

† While a Netscape engineer designed previous versions of SSL, Paul Kocher, a well-regarded cryptographer, designed Version 3 of the protocol, and it has subsequently seen significant review, especially during the standardization process that led to TLS.

Unfortunately, people commonly configure their clients and servers to handle both versions of the protocol. Don't do that. Support only SSLv3 and TLS, to whatever degree possible. Note that clients can't really support TLS only, because TLS implementations are supposed to be able to speak SSLv3. If you wish to use only TLS in a client, you must connect then terminate the connection if the server chooses SSLv3.

As we mentioned when discussing different types of cryptographic algorithms, you should also avoid small key lengths and, to a lesser degree, algorithms that aren't well regarded. 40-bit keys are never secure and neither is 56-bit DES. Nonetheless, it's common to see servers that support only these weak keys, due to old U.S. export regulations that no longer apply.

As for individual algorithm choices in SSL, RC4 and 3DES are both excellent solutions. RC4 is much faster, and 3DES is more conservative. Soon, TLS will be standardizing on AES, at which time this will be widely regarded as a good choice.

Note that the server generally picks a cipher based on a list of supported ciphers that the client presents. We recommend supporting only strong ciphers in the server, where feasible. In other cases, make sure to prefer the strongest cipher the client offers. We discuss cipher selection in detail in Chapter 5.

What SSL Doesn't Do Well

SSL is a great general-purpose algorithm for securing network connections. So far, we've seen the important risks with SSL that you must avoid. Here, we'll look at those things people would like SSL to do, even though it doesn't really do them well (or at all).

Other Transport Layer Protocols

SSL works well with TCP/IP. However, it doesn't work at all with transport layer protocols that are not connection-oriented, such as UDP and IPX. There's not really a way to make it work for such protocols, either. Secure encryption of protocols in which order and reliability are not ensured is a challenge, and is outside the scope of SSL. We do outline solutions for encrypting UDP traffic in Chapter 6.

Non-Repudiation

Let's say that Alice and Bob are communicating over SSL. Alice may receive a message from Bob that she would like to show to Charlie, and she would like to prove that she received the message from Bob. If that was possible, the message would be *non-repudiated*, meaning that Bob cannot deny that he sent the message. For example, Alice may receive a receipt for a product, and wish to demonstrate that she purchased the product for tax purposes.

SSL has no support for non-repudiation. However, it is simple to add on top of SSL, if both Alice and Bob have well-established certificates. In such a case, they can sign each message before SSL-encrypting it. Of course, in such a situation, if Bob wishes to have a message he can repudiate, he just attaches an invalid signature. In such a case, Alice should refuse further communications.

In Chapter 10, we discuss how to sign encrypted messages using S/MIME. This same technique can be used for sending messages over SSL by signing the data before sending it. Alternatively, S/MIME messages could simply be sent over an SSL connection to achieve the same result.

Protection Against Software Flaws

Sometimes SSL fails to secure an application because of a fundamental security flaw in the application itself, not because of any actual problem in SSL's design. That is, SSL doesn't protect against buffer overflows, race conditions, protocol errors, or any other design or implementation flaws in the application that uses SSL.

Even though there are many common risks when deploying SSL, those risks are often minor compared to the gaping holes in software design and implementation. Attackers will tend to target the weakest link, and SSL is often not the weakest link.

Developers should thoroughly educate themselves on building secure software. For administrators deploying other people's software, try to use well-regarded software if you have any option whatsoever.

General-Purpose Data Security

SSL can protect data in transit on a live connection, but it provides no facilities for protecting data before it is sent, or after it arrives at its destination. Additionally, if there is no active connection, SSL can do nothing. For any other data security needs, other solutions are necessary.

OpenSSL Basics

Now that you have a good understanding of cryptography basics, and have seen the SSL protocol at a high level (warts and all), it's time to look specifically at the OpenSSL library. OpenSSL is a derived work from SSLeay. SSLeay was originally written by Eric A. Young and Tim J. Hudson beginning in 1995. In December 1998, development of SSLeay ceased, and the first version of OpenSSL was released as 0.9.1c, using SSLeay 0.9.1b (which was never actually released) as its starting point. OpenSSL is essentially two tools in one: a cryptography library and an SSL toolkit.

The SSL library provides an implementation of all versions of the SSL protocol, including TLSv1. The cryptography library provides the most popular algorithms for symmetric key and public key cryptography, hash algorithms, and message digests. It

also provides a pseudorandom number generator, and support for manipulating common certificate formats and managing key material. There are also general-purpose helper libraries for buffer manipulation and manipulation of arbitrary precision numbers. Additionally, OpenSSL supports most common cryptographic acceleration hardware (prior to Version 0.9.7, forthcoming as of this writing, hardware support is available only by downloading the separate "engine" release).

OpenSSL is the only free, full-featured SSL implementation currently available for use with the C and C++ programming languages. It works across every major platform, including all Unix OSs and all common versions of Microsoft Windows.

OpenSSL is available for download in source form from *http://www.openssl.org/*. Detailed installation instructions for a variety of platforms, including Unix, Windows, Mac OS (versions prior to Mac OS X), and OpenVMS are included in the source distribution. If you're installing on Mac OS X, you should follow the Unix instructions.* The instructions for Mac OS and OpenVMS are very specific for their respective platforms, so we'll not discuss them here. Instead, we recommend that you read and follow the instructions included with the source distribution carefully.

Installations on Unix and Windows have similar requirements; they both require Perl and a C compiler. On Windows systems, Borland C++, Visual C++, and the GNU C compilers are supported. If you want to use the assembly language optimizations on Windows, you'll also need either MASM or NASM. The details of how to build on Windows vary depending on which compiler you're using and whether you're using the assembly language optimizations. We recommend that you refer to the included installation instructions for full details.

The process of building OpenSSL on Unix and Windows systems involves first running a configuration script that is included in the distribution. The configuration script examines the environment on which it's running in to determine what libraries and options are available. Using that information, it builds the `make` scripts. On Unix systems, the configuration script is named *config*; it figures some Unix-specific parameters and then runs the *Configure* script, which is written in Perl. On Windows systems, *Configure* is run directly. Example 1-1 shows the basic steps necessary to build on a Unix system.

Example 1-1. Building and installing OpenSSL on a Unix system

```
$ ./config
$ make
$ make test # This step is optional.
$ su  # You need to be root to "make install"
# make install
```

* OS X comes with the OpenSSL library preinstalled, but it is usually not the most current version. Additionally, if you are a developer, the OpenSSL header files are most likely not installed.

Once the configuration script has been run, the source is ready to be compiled. This is normally achieved by running the `make` program. If you're building on Windows with Visual C++, you'll need to use the `nmake` program. On Unix systems, once the build is complete, some optional tests can be run to ensure that the library was built properly. This is done by running `make test`, as shown in Example 1-1.

When the library is finally built and optionally tested, it's ready to be installed. On Unix systems, this is done by running `make` again and specifying a target of `install`. On Windows systems, there is no install process, per se. You'll need to create directories for the header files, import libraries, dynamic load libraries, and the command-line tool. You can place the files anywhere you like, but you should make sure that you put the DLLs and command-line tool into a directory that is in your path.

Securing Third-Party Software

While much of this book focuses on how to use the OpenSSL API to add security to your own applications, you'll often want to use OpenSSL to secure other people's applications. Many applications are already built to support OpenSSL. For example, OpenSSH uses the OpenSSL cryptography foundation extensively, and requires the library to be present before it can compile. In this particular case, the normal process of installing the software will take care of all the details, as long as you have a version of OpenSSL installed in a well-known place on the system. Otherwise, you can explicitly specify the location of OpenSSL when configuring the software.

OpenSSH is special, because it requires OpenSSL to function. However, many other software packages can support OpenSSL as an option. MySQL is a great example. Simply configure the package with two options, `--with-openssl` and `--with-vio`, and the package will build with SSL support.*

Sometimes it would be nice to use SSL for encrypting arbitrary protocols without actually modifying the source code implementing the protocol. For example, you may have a preferred POP3 implementation that does not support SSL. You'd like to make an SSL-enabled version available, but you have no desire to hack OpenSSL into the code.

In most cases, you can use Stunnel (*http://www.stunnel.org/*) to SSL-enable arbitrary protocols, which it does by proxying. Stunnel in and of itself is not a complete tool—it requires OpenSSL to run.

You can use Stunnel to protect HTTP traffic. However, it's generally better to use the web server's preferred SSL solution. For example, Apache's mod_ssl (see *http://www.modssl.org*) is a far better solution for Apache users than Stunnel, because it is far

* By default, MySQL connections are not encrypted, even after compiling with SSL. You have to explicitly state that a particular user connects with SSL. See the MySQL GRANT documentation for details.

more configurable. And, under the hood, mod_ssl also uses the OpenSSL library. The details of mod_ssl are beyond the scope of this book. For more information on this topic, refer to the mod_ssl web site or the book *Apache: The Definitive Guide*, by Ben Laurie and Peter Laurie (O'Reilly).

Server-Side Proxies

Let's say that we want to run SSL-enabled POP3 on the standard port for this (995). If we already have the unencrypted POP3 server running on port 110, we simply put Stunnel on port 995, and tell it to forward connections to port 110 on the loopback interface (so that unencrypted data isn't sent over your local network, just to come back onto the current machine). When SSL-enabled POP3 clients connect to port 995, Stunnel will negotiate the connection, connect itself to the POP3 port, then start decrypting data. When it has data to pass on to the POP3 server, it does so. Similarly, when the POP3 server responds to a client request, it talks with the Stunnel proxy, which encrypts the response, and passes it on to the client. See Figure 1-5 for a graphical overview of the process.

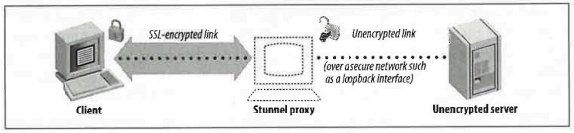

Figure 1-5. Stunnel proxies

To use Stunnel on the server side, you must install a valid server certificate and private key. An appropriate Certification Authority should sign the certificate. You can generate your own credentials using OpenSSL. That process is covered in Chapter 3.

These server credentials will need to be made available to Stunnel. Often, the correct location of these credentials will be hardcoded into the Stunnel binary. If not, you can specify their location on the command line.

Assuming the POP3 server is already running, here is how you would run Stunnel from the command line to implement the above scenario (assuming that you're running as root, which is necessary for binding to low ports on Unix machines):

```
# stunnel -d 995 -r 127.0.0.1:110
```

The `-d` flag specifies that Stunnel should run as a proxy in daemon mode on the specified port (you can also specify the IP address on which to bind; the default is all IPs on the machine). The `-r` flag specifies the location of the service to which Stunnel will proxy. In this case, we specifically mention the loopback address to avoid exposing unencrypted traffic to other hosts on the same local network. Optionally, we could hide the port from external eyes using a firewall.

The location of the certificate file can be specified with the `-p` flag, if necessary. If your machine's services file contains entries for the POP3 and the Secure POP3 protocol, you can also run Stunnel like this:

```
# stunnel -d pop3s -r 127.0.0.1:pop3
```

You can also run Stunnel from inetd. However, this is generally not desirable, because you forego the efficiency benefits of session caching. If you're running on Windows, Stunnel is available as a precompiled binary, and can be easily launched from a DOS-style batch file. See the Stunnel FAQ (*http://www.stunnel.org/faq*) for more details.

Unfortunately, Stunnel can't protect all the services you might want to run. First, it can protect only TCP connections, not UDP connections. Second, it can't really protect protocols like FTP that use out-of-band connections. The FTP daemon can bind to arbitrary ports, and there's no good way to have Stunnel detect it. Also, note that some clients that support SSL-enabled versions of a protocol will expect to negotiate SSL as an option. In such cases, the client won't be able to communicate with the Stunnel proxy, unless it goes through an SSL proxy on the client end as well.

Since Stunnel will proxy to whatever address you tell it to use, you can certainly proxy to services running on other machines. You can use this ability to offload the cost of establishing SSL connections to a machine by itself, providing a cost-effective way of accelerating SSL. In such a scenario, the unencrypted server should be connected only to the SSL proxy by a crossover cable, and should be connected to no other machines. That way, the unencrypted data won't be visible to other machines on your network, even if they are compromised. If you have a load balancer, you can handle even more SSL connections by installing additional proxies (see Figure 1-6). For most applications, a single server is sufficient to handle the unencrypted load.

The biggest problem with using Stunnel as a proxy is that IP header information that would normally be available to the server isn't. In particular, the server may log IP addresses with each transaction. Since the server is actually talking to the proxy, from the server's point of view, every single connection will appear to come from the proxy's IP address. Stunnel provides a limited solution to this problem. If the secure port is on a Linux machine, then the Stunnel process can be configured to rewrite the IP headers, thus providing transparent proxying. Simply adding the `-T` flag to the command line does this. For transparent proxying to work this way, the client's default route to the unencrypted server must go through the proxy machine, and the route cannot go through the loopback interface.

Stunnel can be configured to log connections to a file by specifying the `-o` flag and a filename. That at least allows you to get information about connecting IP addresses (which should never be used for security purposes anyway, since they are easy to forge), even when transparent proxying is not an option.

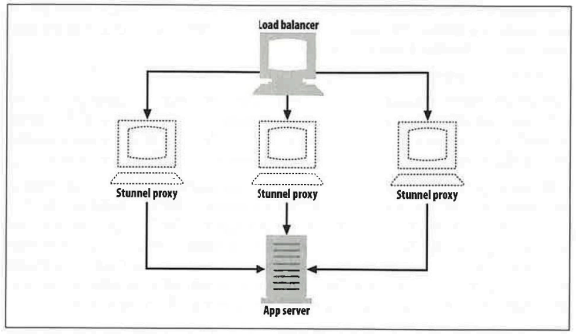

Figure 1-6. Load balancing with Stunnel for cryptographic acceleration

Client-Side Proxies

Stunnel can also be used to connect clients that are SSL-unaware with servers that do speak the protocol. Setting up a client-side proxy is a bit more work than setting up a server-side proxy because, while clients are usually authenticated using some sort of password mechanism, servers are authenticated primarily using cryptographic certificates. You can set up the client not to authenticate, but if you do so, be warned that man-in-the-middle attacks will be easy to perform. Unauthenticating client proxies only buys you security against the most naïve eavesdropping attacks, but is still better than no protection at all.

Let's start with a case in which we are not yet validating certificates. Let's say that we'd like to connect to Amazon.com's SSL-enabled web server, running on port 443 on *www.amazon.com*. First, we can interactively test the connection by running Stunnel in client mode (specified by the -c flag):

```
$ stunnel -c -r www.amazon.com:443
```

Stunnel silently connects. We type in an HTTP request, and get back the appropriate response. For example:

```
GET /
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>302 Found</TITLE>
</HEAD><BODY>
<H1>Found</H1>
The document has moved <A HREF="http://www.amazon.com/">here</A>.<P>
</BODY></HTML>
```


After sending its response, the server closes the connection.

As you can see, we can talk with the SSL-enabled web server running on Amazon.com, yet the SSL handling is completely transparent from our point of view.

Running Stunnel in interactive mode is useful for the purposes of debugging. However, interactive mode is not practical for use with arbitrary clients. Let's say we wish to point an SSL-unaware POP3 client at an SSL-enabled POP3 server running on *mail.example.com*. On the machine running the client, we would like to set up a proxy that only accepts connections from the local machine, and then makes connections on behalf of the local machine to the SSL-enabled server. We can easily do that with the following command:

```
# stunnel -c -r mail.example.com:pop3s -d 127.0.0.1:pop3
```

This command sets up a proxy on the local machine that does what we want it to. Now we can simply point our mail client to our loopback interface, and we will magically connect to the intended SSL-enabled POP3 server (assuming no man-in-the-middle attacks).

Note that the above command will work only if you have permission to bind to the POP3 port locally. If that is an issue, and your POP client can connect to servers on arbitrary ports, the problem is easy to fix. Otherwise, you'll need to grant the proxy process root privileges, or find a new client. Root privileges pose a big risk, because there may be an obscure security bug in Stunnel that would allow data passing through the proxy to gain root privileges. If you do choose to grant the proxy root privileges, on most operating systems you should probably run the proxy as root, and then use the -s flag to specify a username to switch to after the port is bound. You might consider making the binary setuid—but you shouldn't, because you would then let any user bind to privileged ports as long as he can run the Stunnel binary.

As we mentioned previously, you should always have client proxies perform certificate validation. To use certificate validation, you must specify where on the client machine valid CA certificates live, and you must specify the level of validation you want. We recommend maximum validation (level 3), and we think you should completely stay away from level 1, since it offers no real validation. Here's an extension of the above example that takes into account certificate validation:

```
# stunnel -c -r mail.example.com:pop3s -d 127.0.0.1:pop3 -A /etc/ca_certs -v 2
```

The file */etc/ca_certs* stores a list of trusted CA certificates (see Chapter 3 for more information on obtaining such certificates). Unfortunately, Stunnel doesn't support validation based on domain-name matching. If you wish to restrict valid servers to a small set (usually a very good idea), you can use validation level 3 (the maximum), and place the known certificates in a directory of their own. The certificate's filename must be the hash value of the certificate's subjected (see the -hash option to the x509 command in Chapter 2 to find out how to generate this value), with a ".0"

file extension. Additionally, you use the -a flag to specify where valid server certificates live. For example:

```
# stunnel -c -r mail.example.com:pop3s -d 127.0.0.1:pop3 -A /etc/ca_certs -a
/etc/server_certs -v 3
```

Again, we talk more about certificate formats in Chapter 3.

As with server-side SSL proxies, there are some situations in which client-side use of Stunnel isn't appropriate. Once again, it doesn't make sense to use Stunnel in a UDP-based environment or with a protocol that makes out-of-band connections. Additionally, some servers that support SSL expect to negotiate whether or not to use it. These servers won't understand a connection that is encrypted with SSL from start to finish. Such negotiation is especially popular with SSL-enabled SMTP servers.

Stunnel has support for negotiating some of the more common protocols. To use that support, invoke Stunnel in the same way as in the previous client-side example, but add the -n argument, which takes a single argument (the name of the protocol). Currently, SSL supports SMTP, POP3, and NNTP. For example, to connect to a secure SMTP server over SSL, use the command:

```
# stunnel -c -r mail.example.com:smtp -d 127.0.0.1:smtp -A /etc/ca_certs -a /etc/
server_certs -v 3 -n smtp
```

Unfortunately, as of this writing, Stunnel doesn't support any other protocols for which SSL is a negotiated option, most notably SSL-TELNET.

CHAPTER 2

Command-Line Interface

OpenSSL is primarily a library that is used by developers to include support for strong cryptography in their programs, but it is also a tool that provides access to much of its functionality from the command line. The command-line tool makes it easy to perform common operations, such as computing the MD5 hash of a file's contents. What's more, the command-line tool provides the ability to access much of OpenSSL's higher-level functionality from shell scripts on Unix or batch files on Windows. It also provides a simple interface for languages that do not have native SSL bindings, but can run shell commands.

There's no question that the command-line tool can seem quite complex to the uninitiated. It sports a large set of commands, and even larger sets of options that can be used to further refine and control those commands. OpenSSL does come with some documentation that covers most of the available commands and options supported by the command-line tool, but even that documentation can seem intimidating. Indeed, when you're trying to discover the magical incantation to create a self-signed certificate, the documentation provided with OpenSSL does not provide an intuitive way to go about finding that information, even though it is in fact buried in there.

This chapter contains an overview of the command-line tool, providing some basic background information that will help make some sense of how the tool's command structure is organized. We'll also provide a high-level overview of how to accomplish many common tasks, including using message digests, symmetric ciphers, and public key cryptography. The Appendix contains a reference for the commands that the command-line tool supports.

We will refer to the command-line tool throughout this book, and, in some instances, we also provide examples that are more complex than what we've included in this chapter. In particular, Chapter 3 makes extensive use of the command-line tool.