Securing Sockets with OpenSSL (part 1)

The Internet is not for the weak-hearted programmer and designer. Today more than ever you as a professional have to outwit and out-think some of the brightest and most devious minds. In fact people, companies, and governments have set up public and private agencies for tracking and tracing “cyber-perps” (or crackers) a new kind of crime that neither the agent nor the cracker may ever see each other’s face. In spite of the gotchas, network programming is hot and will continue to be so for years to come.

The whole arena of cyber-this and cyber-that revolves around the Internet, and the Internet uses TCP/IP as its mainstay protocol. Earlier articles describe TCP (or streaming protocols) and UDP (or datagram protocols). These protocols provide the brick and mortar of the Internet arena. On top of the brick and mortar lie several additional protocols that provide authentication, session management, and virtual services.

This is the first part of a two part series, and it covers the issues and terminology regarding message security. The second part shows you examples how to write secure clients and servers, leveraging the concepts from an earlier article on TCP programming called “Writing a C-based Client/Server”.

Differentiating the Authorities

The higher network functions depend usually on TCP’s reliable communication channel. Two exceptions are Network File System (NFS) which uses the faster datagram protocols and Remote Procedure Calls (RPCs) which let you choose either TCP or UDP. But, higher level or application layer tools use streaming connections (see Figure 1, right). The reason is clear: TCP provides reliability, unlimited message size, and an implicit client authentication. This authentication is due to the fact that the two computers are virtually connected in a session; as long as the connection is uninterrupted, a program usually can assume that the messages it’s getting come from the same program that started the session. That’s a big assumption.

Many of the application layer programs use some form of security. Security has three main pillars: authentication, authorization, and certification. Authentication is basically: “I am so-and-so” and typically uses a username/password-style. If the user knows the name and password, most systems assume plausible authentication.

Authorization associates the user with certain privileges. The authorities file on some operating system describes the permissions that an active (online) and inactive (offline) user may have. Linux uses user/group/world permissions, and Windows uses Access Control Lists (ACLs). When you use telnet, an application-layer program, it ties your authentication with your authorization. In other words, when you log in, the server assigns your authorities as part of authentication.
Both authentication and authorization are pretty straightforward. Certification, on the other hand, requires a little more explanation. As noted above, some systems assume that just because the client program provides username/password or is still connected to the server, the client is really who they say they are. This is a dangerous programming assumption to make if writing transaction protocols for banking transfers.

Certification requires a trusted third party that holds information about each subscriber. This information, or certificate, affirms that the holder is really who they say they are (and if the program needs to check it they can through the third party). The certificate holds several pieces of information: holder name, issuer, expiration date, etc. This certificate is signed, a process similar to watermarking of checks which demonstrates authenticity.

The certification process participates centrally in the secure connection. Each piece of the certificate plays a role in how the client and server set up their trust, or secure link.

Defining the Privacy Problem

Network security has a particular problem: an open (insecure) channel is a back door for cracking an authenticated link. For instance, programs never should send passwords unencrypted. The fact is that passing information to/from sites in a public setting like the Internet is commonplace. And the Internet is the last place for a private conversation.

The Internet’s power causes the privacy problem. Built in the Cold War paranoia, it was designed to adapt quickly to lost segments, paths, and networks. This means that a message can go in and out of the network’s equivalent of a no-man’s land. In this region, networking perps snoop messages and try to take over a connection (hijack). If a message looks interesting, the cracker may investigate more and even try to entice the local routers to direct traffic towards his/her segment, making it easier to see more data. If every connection were directly connected to the desired server, the problem would be not as severe. Instead, you can add some level of privacy to your programs with encryption.

Encryption is not a 100% silver bullet and will only provide a degree of privacy. Different encryption algorithms (ciphers) scramble the data to apparent randomness, making snooping difficult. However, as technology progresses and as computers grow in computing power, the snoopers can apply more resources to crack open the critical messages.

You have a wide variety of ciphers to work with, and depending on your country’s laws, you may use them internationally. Ciphers come in two basic forms:

- **destructive (lossy)** - Data going in does not come out.
- **nondestructive (lossless)** - The data is recoverable.

The telnet program, for example, uses a lossy encryption for authentication because it only needs to check against a known value (the password). Unknown values (the store does not know your credit card number until you tell them) require a lossless cipher. Lossless ciphers fall in two groups, public and symmetric (AKA secret or private) key encryption.

Symmetric key encryption uses a reversible algorithm that scrambles and unscrambles using the same key. The sending and receiving hosts share the key. Of course, the no one else can know the key (hence secret); otherwise, someone could crack the code and get to the data. Symmetric keys serve best after the two host computers have completed the certification and are ready to perform transactions. They are relatively fast and very secure.
Public key encryption, on the other hand, is like a special drop box: it uses two keys, one for encryption and the other for decryption. The server hands out the encryption (public) key, and the clients use it to encrypt sensitive data for the server. The server accepts the data and decrypts the data with the decryption (private) key. This is a very clever and useful tool for the Internet – everyone can get a copy of the public key, but only the issuer can decrypt the messages. However, public ciphers are about 10x slower than symmetric key ciphers and are less than ¼ as secure.

Network privacy continues to drive the technology. The speed, reliability, security, availability, and portability of the ciphers force scientists to more creative solutions. The networking industry recognizes that privacy is crucial for their connected-ness, and if they want to remain connected, they must adopt technologies that ensure their data retains integrity and that protect their internal network.

Of course, the client also wants his/her information protected but often leaves the privacy protection in the hands of the service provider. This has led to some data privacy and usage statements you may have seen on the Internet. You, as the designer, must work within the confines of good sense to protect the users’ privacy (and consulting your corporate ethics and security groups may give added insight). One bad example was when one company’s site asked for the user’s social security number (SSN), but they chose not to ask for the email address. They felt that the email address was more personal than the SSN. Clearly, legality is an important issue as well.

Certifying the Participants

The service provider can only legitimately protect the clients’ information if it is what it claims to be. As described above, establishing the participant’s authenticity with certificates is currently the preferred method in establishing secure channels.

Both the client and the server can pass certificates to each other, but more servers more often do not expect a certificate from the client. E-commerce (through an Internet storefront) usually certifies the server, while the client remains uncertified. Banking and stock brokering, alternatively, requires client certification. The reason stems from the question: “Does the peer need to see confidential information?” If the client needs to see bank records (clearly confidential), then it needs to certify itself. On the other hand, the Internet storefront does not share any confidential information, but since it requests a credit card (confidential), the client must expect a certificate.

Doubling the Handshake

The Secure Socket Layer (SSL) uses certificates as part of the handshake between the client and server. When the client connects to the server, they exchange a series of messages that do much more than just authenticate each other. The Transaction Control Protocol (TCP) uses a handshake to begin and end a streaming conversation between the two host computers (see the “Writing a C-based Client/Server” article for more detail). SSL adds many more messages on top of TCP’s simple handshake.

During the SSL handshake, the client and server perform several steps. These steps check and cross check so that the connection is as secure as technically feasible (see Figure 2, below). The first steps (“Hello”) make sure that the client and server are talking the same language. Secure sockets come in several flavors, SSL 2.0, SSL 3.0, TLS 1.0, and PCT 1.0.
Securing Sockets with OpenSSL

The client can optionally send its own certificate in the first part of the handshaking dialog. After exchanging the *Hello* messages, the client and server exchange public keys and begin the negotiation process. During this process they select the ciphers and secret keys.

After the two hosts complete the handshake, they switch to using the faster, more reliable symmetric ciphers. The programs then send and receive messages as usual.

One implementation of SSL/TLS is OpenSSL. The way the client and server interface is very simple, once both programs complete the secure channel. In fact, the only change between regular socket programming and secure socket programming is the initial overhead and the new `send()`/`recv()` library calls. You can easily convert many C-socket programs with a search and replace.

From Theory to Application

TCP provides the needed reliability to the SSL API. SSL adds more handshaking so that both the client and server can use the best ciphers and ensure that you have a private link. Part 2 of this article describes how to use OpenSSL, a production-ready API, for connecting with secure clients and servers.

If you would like to know more about sockets programming and SSL, pick up a copy of *Linux Socket Programming* from SAMS. You can also get the complete source listings for the book and these articles at [http://www.cs.utah.edu/~swalton](http://www.cs.utah.edu/~swalton).