

An Introduction to OpenSSL, Part Four: The SSL and TLS Protocols

Holt Sorenson 2001-10-03

An Introduction OpenSSL, Part Four: The SSL and TLS Protocols

by Holt Sorenson

last updated October 3, 2001

This article completes the four-part series on OpenSSL, a library, written in the C programming language, that provides routines for cryptographic primitives utilized in implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. OpenSSL also includes routines for implementing the SSL and TLS protocols. An application called `openssl`, which provides a command line interface to the library's routines, is also part of the distribution. This article will provide some background on the SSL protocol and its relationship to TLS. It will also discuss TLS in depth, and show how users can use OpenSSL to set up and test TLS/SSL connections.

An Introduction to Protocols Using a Wanna-be Protocol

A protocol is a detailed procedure that is agreed upon by two or more parties that facilitates those parties' communication. A cryptographic protocol can consist of cryptographic algorithms, key set-up, random number generation and other methods needed to facilitate the protocol. Previous articles in this series (see the relevant links box at the end of this article) have shown how to use OpenSSL's command line interface to manipulate cryptographic primitives. The following section will introduce a fairly simple protocol that is composed of steps that illustrate how PGP uses cryptography to complete RSA encryption and signing. Exploring these steps with OpenSSL commands is left as an exercise for the reader. This bonded and disciplined regimen is called, for the sake of this discussion, the "I wanna be a real crypto protocol when I grow up protocol" or IWBARCPWIGUP. IWBARCPWIGUP is sometimes called the twelve-step protocol.

(Writing real crypto protocols takes time, effort, and significant amounts of peer review. This protocol hasn't had any of that. It is presented here for readers to practice using OpenSSL, not to protect data. Please don't use it for any purpose other than practice and familiarization. Hopefully, the protocol's name will drive home this point.)

When the protocol's steps are completed, the two parties will have exchanged one message each with each other. The medium the parties are communicating across is known to be inherently insecure. By using the protocol the parties have a reasonable assurance that the integrity of the messages will not have been compromised, that the confidentiality of the messages will have been

preserved, and that the parties will be able to verify that the messages were created by the expected originator (authentication).

For the sake of this discussion, let's call the two parties using the IWBARCPWIGUP protocol Alice and Bob. Alice and Bob generate 1024 bit RSA key pairs and exchange the public keys. Alice and Bob create messages for one another and include information that indicates who the message is for and who it is from. Pseudonyms are great if they're feeling paranoid. They consult the IWBARCPWIGUP specification and it outlines the following procedure:

Alice's steps:

- 1) Using SHA1, make a hash of the plaintext.
- 2) Sign the hash generated in step 1 with Alice's private RSA key and the RSA asymmetric cipher.
- 3) Generate a 128-bit key from cryptographic randomness.
- 4) Append the signature generated in step 2 to the plaintext (the message to be sent) and encrypt the result using the Blowfish symmetric cipher and the 128-bit key generated in step 1.
- 5) Encrypt the 128-bit key used with Blowfish in step 3 using Bob's public RSA key and the RSA asymmetric cipher.
- 6) Send the ciphertext from steps 4 and 5 to Bob.
- 7) Wait for Bob's message.
- 8) Decrypt the ciphertext from Bob's step 5 (explained below) using the RSA symmetric cipher and Alice's private key to recover the 128-bit key generated in Bob's step 3.
- 9) Decrypt the ciphertext generated in Bob's step 4 using Blowfish and the key recovered in step 8 to recover the plaintext message and the RSA signature of the SHA1 hash.
- 10) Detach the signature from the plaintext message and hash the plaintext message with SHA1.
- 11) Verify the signature of the SHA1 hash using the RSA symmetric cipher and Bob's public key. If the verification is successful, then Bob is the originator of this message. One cool property of the RSA symmetric cipher is that signature verification recovers the information signed in Bob's step

2. Since this recovers the SHA1 hash that Bob created in Bob's step 1, we can compare the recovered hash with the result of step 10. If these SHA1 hashes are the same, the message's integrity has been preserved. If either the signature doesn't verify or the hash doesn't match, do not continue the IWBARCPWIGUP; Execute the abort protocol.

12) Read the message.

Bob's Steps

Bob uses the same steps as Alice but switches names as is appropriate as he follows the protocol. In brief, the protocol uses steps 3-5 to preserve the confidentiality of the message, steps 1-2 and 10-11 to preserve the integrity of the message, and steps 2 and 11 to provide authentication of the party sending the message.

To reiterate: please don't forget this is a protocol for readers to use as they experiment with OpenSSL's crypto functions, not a protocol for real life use. Protocols that are sufficient for securing infrastructure have been created and studied by groups of professionals who have put time, sweat, blood, and tears (and lots of flame messages) into the protocol creation processes. Don't use this protocol for anything you care about. Use something like OpenPGP [[Callas1998](#)].

The TLS/SSL Protocol

SSL is a protocol that utilizes cryptographic primitives to provide confidentiality, integrity, and authentication during a persistent session between two hosts in a hostile environment. SSL was initially designed by Netscape at the beginning of 1994 [[SSLHistory](#)] to protect HTTP [[Berners-Lee1999](#)] sessions involved in electronic commerce transactions. SSL has evolved over three-point-one (more like four, but that's another story) versions into TLS (Transport Layer Security) [[Rescorla2000_1](#)]. TLS 1.0 adds some modifications to SSL 3.0. These modifications are minor enough that the [TLS RFC](#) states: "TLS version 1.0 and SSL 3.0 are very similar; thus, supporting both is easy." Mel and Baker elaborate on that statement by saying there are "some minor changes in HMAC calculations, cipher suite support, and pseudo-random number calculations. You can think of TLS as SSL v3.1" [[Mel2000_1](#)]. Eric Rescorla [[Rescorla2000_2](#)] points out that TLS uses the HMAC standard and that SSLv3 uses an earlier version of HMAC (see [RFC 2104](#) and [RFC 2202](#) for more details.) Current versions of most web browsers and servers implement the TLS protocol.

If one of the peers (the client or server) lacks support for the TLS protocol, TLS provides a

mechanism for rolling back to the SSL v3.0 protocol as TLS is not compatible with SSL v3.0 [[Dierks1999_3](#)].

Since the trend with secure web sites seems to be towards using TLS_RSA_WITH_RC4_CBC_MD5, we will describe a key exchange that progresses with that cipher suite and assume that the client isn't authenticating with a certificate. We also assume that both the client and server are capable of using 128-bit keys for (US domestic) symmetric encryption [[KeyLengthCite](#)], and that the server's x.509v3 certificate is a signed 1024 bit RSA key.

- 1) The TLS protocol begins by the client initiating the TLS handshake protocol with a hello message that includes the protocol version, session id, cipher suite, compression method, and a random number [[Dierks1999_4](#)].
- 2) The server responds with a hello message that includes the protocol version, session id, cipher suite, compression method, and a random number.
- 3) The server then sends its certificate and a message indicating it is finished with the hello message exchange.
- 4) The client responds with a key exchange message. The exact steps for key exchange and the contents of the key exchange message are dependent on the cipher suite previously negotiated. In the case of TLS_RSA_WITH_RC4_CBC_MD5, the client generates a 48-byte premaster secret and encrypts it using the server's RSA public key contained in the certificate that the server sent in step three. The 48-byte premaster secret is composed of 2 bytes that identify the client version and 46 random bytes.
- 5) The server decrypts the key exchange message with the server's private RSA key and verifies that the first two bytes of the premaster secret are the client version that is the same as the client version included in the initial hello message of the TLS handshake protocol (step one) [[Dierks1999_5](#)].
- 6) Both the client and server use the random numbers exchanged as parts of the hello messages in steps one and two and the premaster secret with a pseudo random function (PRF) to generate the master secret [[Dierks1999_6](#)]. The master secret is 48 bytes long. This gives us 384 bits, which is equally divided into three 128-bit keys. One of these keys is used for the MAC (keyed MD5), one for the symmetric encryption key (RC4), and one as the initialization vector for the symmetric algorithm.

7) The client and server send 'handshake finished' messages to one another with the now set up cipher suite. The 'handshake finished' messages are protected by the set of keys created generated from the master secret in step six.

8) Assuming that neither the client nor the server request that the now negotiated session keys be expired and new keys be negotiated, the session keys, based on the master secret, are used for the remainder of the TLS session.

Securely Implementing SSL

One of the more critical objects that need to be protected in TLS is the server's private key. The server offers its RSA public key to the client and the client encrypts the pre master secret to that public key. The server decrypts the pre master secret with its private key. The server and client generate the symmetric cipher key, initialization vector, and MAC key from the pre master secret and the random values previously exchanged. The random values were exchanged in the clear, so all the attacker needs is the pre master secret to compute the same keys that the client and server compute. Once the attacker computes the same keys, they can passively (I.E. eavesdrop) or actively (I.E. hijack) attack the SSL session between the client and the server. To compromise the pre master secret, the attacker needs the server's private key.

For other possible attacks on TLS/SSL consult [\[Rescorla2000_4\]](#).

The private key needs to be generated in a secure manner with a cryptographically strong random number generator (consult the randomness section of the first article in this series). The private key should be generated on the server that will be engaging in the cryptographic operations. The private key should never leave the server. The private key should only be usable by the users, groups, or processes that will be engaging in the cryptographic operations. When the private key is decommissioned, it should be destroyed [\[DDCite\]](#) This prevents any attackers who may have been recording sessions from recovering the private key from the server. If the private key is ever stored on a medium not under the direct control of the server, the detached media should be treated with the same deference that the security model requires of the server and should be destroyed when it is no longer needed.

Using the OpenSSL application for TLS/SSL

Now that the theory is out of the way, let's play around a bit with the protocol. First we need to generate a certificate for the server to use. We can do this by running the command `openssl req`

as follows:

```
openssl req -newkey rsa:1024 -x509 -nodes -keyout test_key.pem -new -out test_cert.pem
```

Refer to the third [article](#) in this series for more information.

Assuming that you have no processes bound to port 4433, you can now run `s_server`:

```
openssl s_server -cert test_cert.pem -key test_key.pem -state -debug
```

and begin connecting to it with `s_client`:

```
openssl s_client -debug -state
```

You can use (typically `^C`) or (typically `^D`) to finish a session. The server will continue to accept on the port until you stop it, even if you disconnect the client. You can also use OpenSSL to test web servers or other protocols that are commonly used over TLS/SSL. If IMAP over SSL is deployed at your site, you can use OpenSSL to connect so you can test IMAP by hand using:

```
openssl s_client -debug -state -connect :993
```

If the remote host forces TLS, you will have to include the **-tls1** option.

Throughout this series of articles, we have discussed several technologies that constitute the TLS/SSL protocol. It is worth reiterating that all of them need to be deployed and utilized in a secure manner. Security expert Carl Ellison tells a story about chaining his bike to a wooden gate with a "hefty padlock" that "cost a fair amount." His neighbor knocked on his door one morning with Carl's bike pack in hand. Carl didn't immediately recognize it, because his "bike was safely locked up" and this familiar looking pack was certainly attached to it [\[Ellison2001\]](#).

Security systems are like that padlock. They are designed to mitigate specific threats when properly deployed. Unfortunately, humans often neglect to design and deploy systems that are holistically secure. They make the mistake of relying on components of the system that are certainly secure, but they are negligent in analyzing all the components of the system and their interfaces. The problem is exacerbated when the system isn't built only upon components, but is also composed of systems. Some have illustrated this point by likening this process to setting up an asset protecting tent stake in a gargantuan field and expecting that your adversary will run directly into it. Many times the weakest link in the chain security your bicycle is the wooden gate.

Attackers don't work within the design specifications of a system; they make up their own rules. If there is a path of compromise available to your attacker, it doesn't matter how many bits your cryptosystem uses or how much peer review the algorithms and protocols have had; the game is over and you lose.

Check out the [further study section](#) following the bibliography for more learning opportunities.

Holt Sorenson works for Counterpane Internet Security where he wrangles tuples of bits so his colleagues can get their work done. He's always serious and never jokes or laughs. When he is not surgically attached to computers he likes to hang out with his family and engage in frivolous pursuits that purportedly keep him out of trouble.

Relevant Links

["An Introduction to OpenSSL, Part One: Cryptographic Primitives"](#)

Holt Sorenson

["An Introduction to OpenSSL, Part Two: Cryptographic Primitives Continued"](#)

Holt Sorenson

["An Introduction to OpenSSL, Part Three: Public Key Infrastructure"](#)

Holt Sorenson

[Privacy Statement](#)

Copyright 2006, SecurityFocus