SSL/TLS Protocols

SSL (Secure Socket Layer, developed by Netscape) & TLS (Transport Layer Security, is an IETF standard) are almost the same. They run as a user-level processes on top of TCP/IP.

The Basic Protocol:

{----------------------------------------
Alice                                                                              Bob
I want to talk, ciphers I support, Ra  ————————>
<—————————— certificate, cipher I choose, Rb
choose secret S, compute K= f(S,Ra,Rb):
{S}_{Bob}, \{keyed hash of handshake msgs\} ————————>
compute K= f(S,Ra,Rb):
<—————————— \{keyed hash of handshake msgs\}
<—— data protected with keys derived from K———>
{----------------------------------------

Keys:

- Alice chooses a random number \( S \), known as the pre-master secret.
- It is shuffled with \( Ra \) and \( Rb \) to produce a master secret \( K \).

- \( Ra \) and \( Rb \) are 32 octets long, the first 4 are the UNIX time (seconds since Jan 1, 1970).

This ensures that \( R \)s are always different.

- The master secret is shuffled with the two \( R \)s to produce six (6) keys:

  Three for each side for encryption, integrity, and IV. The three keys used for transmission are known as the write keys
while the three used for receipt are known as the **read keys**
Thus Alice's write keys are Bob's read keys and vice versa.

To ensure that the keyed hash Alice sends is different from the keyed hash Bob sends, Alice include the string "CLNT" and the Bob include "SRVR" in the hash.

Note that Alice has authenticated Bob, but Bob has no idea to whom he's talking.
In SSL it is optional for the server to authenticate the client, if he has a certificate.
Normally the server authenticates the user using:

```
<name, password>
```
sent securely over the ssl connection.

**Session Resumption**

If the server support session resumption, it sends `session_id` for the client.

```
{---------------------------------------
Alice                             Bob
I want to talk, ciphers I support, Ra ____________>
<________ session_id, certificate, cipher I choose, Rb
choose secret S, compute K= f(S,Ra,Rb):
{S}Bob {keyed hash of handshake msgs} ____________>
compute K= f(S,Ra,Rb):
<----------------------{keyed hash of handshake msgs}
<----data protected with keys derived from K----->
{---------------------------------------}
```

**Session resumption if both sides remember the `session_id`:**

```
{---------------------------------------
Alice                             Bob
```
session_id, ciphers, Ra \[\rightarrow\]

\[\leftarrow\] session_id, cipher, Rb, \{keyed hash of msgs\}

\{keyed hash of msgs\} \[\rightarrow\]

\[\leftarrow\] data protected with keys derived from K

\[\equiv\]

Note that they still have to negotiate ciphers, but the pre-master secret S is the same (which is expensive to generate).

### Encrypted Records

<table>
<thead>
<tr>
<th>SeqNum</th>
<th>Header</th>
<th>Data</th>
<th>[→] HMAC [←] integrity key</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>V</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>V</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

\[\leftarrow\] encryption key

<table>
<thead>
<tr>
<th>Header</th>
<th>Data</th>
<th>HMAC</th>
<th>pad</th>
<th>[→] ENCRYPT [←] encryption key</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V</td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

If block cipher is used, the IV is used to encrypt the first record. The final block of each record is used as the IV for the next record.

### Connection Closure

- The sender should send close_notify message to signal the other end that it has no more data to send.

- The purpose is to prevent a truncation attack in which the attacker inserts a TCP FIN segment before the sender is finished sending data forcing the receiver to think that all data has been received.
- If a party receives FIN without first receiving close_notify it must mark the session as not resumable.

---

**HTTP Over SSL - https**

**HTTP:**

HTTP (HyperText Transfer Protocol) is the basic transport protocol of the Web.
Conceptually, it is a very simple protocol. The basic unit of HTTP interaction is the *request/response* pair:

- The client opens a TCP connection to the server and writes the request and
- The server writes back the response and indicates the end of response either with

  a length header or simply be closing the connection:

**Example:**

- **Client Request:**

  ```
  GET / HTTP/1.0
  Connection: Keep-Alive
  Host: www.cs.odu.edu
  ```

- **Server Response:**

  ```
  HTTP/1.0 200 OK
  Content-Length: 1650
  Connection: Keep-Alive
  Content-Type: text/html
  ..........
**URLs:** `<scheme>://<host>[:<port>]/<path>[?<query>]`

Examples:

- `<schems>`: http, default `<port>` 80
- `<schems>`: ftp, default `<port>` 21
- `<schems>`: https, default `<port>` 443

**HTTPS:**

The client makes a connection to the server, Negotiates an SSL connection and Transmits http data over the established secure connection.

- **Reference integrity:**
  
  Match the URL reference to the server's identity with the *CN name* in the server's certificate.

---

**OpenSSL: s_server & s_client**

**Server:**

```bash
% openssl s_server -accept 1234 -cert server_cert.pem -key server_privatekey.pem
```

```bash
% openssl s_server -accept 1234 -cert server_cert.pem -key server_privatekey.pem -WWW
```

*The option (WWW) causes the server to emulate a simple http server.*

```bash
% openssl s_server -accept 1234 -cert server_cert.pem -key server_privatekey.pem -verify 2 -CAfile ca_cert.pem
```
The option `-verify` causes the server to demand a certificate from the client and the depth of the chain should not exceed 2 and the option `-CAfile` specify the trusted certificate.

**Client:**

% openssl s_client -connect localhost:1234 -verify 2 -CAfile ca_cert.pem

% openssl s_client -connect localhost:1234 -verify 2 -CAfile ca_cert.pem -cert client_cert.pem -key client_privatekey.pem

% openssl s_client -connect localhost:1234 -verify 2 -CAfile ca_cert.pem -reconnect

The option `-reconnect` causes 5 connections to the server using the same session ID to test session cashing.

To test the WWW mode of server type: GET /server.pem HTTP/1.0