Chapter 7

WEB Security

By: Bhargavi H Goswami,
Head & Associate Professor,
Department of Computer Science
Garden City College
+91 9426669020
bhargavi.goswami@gardencitycollege.edu
Topic List

1. Web Security Considerations
2. Secure Socket Layer (SSL) and Transport Layer Security (TLS)
3. Secure Electronic Transaction (SET)
4. Recommended Reading and WEB Sites
Overview:

• Secure Socket Layer (SSL) provides security services between TCP and applications that use TCP. The Internet standard version is called Transport Layer Service (TLS).
• SSL/TLS provides confidentiality using symmetric encryption and message integrity using a message authentication code.
• SSL/TLS includes protocol mechanisms to enable two TCP users to determine the security mechanisms and services they will use.
• HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server.
• Secure electronic transaction (SET) is an open encryption and security specification designed to protect credit card transactions on internet.
• Secure Shell (SSH) provides secure remote logon and other secure client/server facilities.
Web Security Considerations:

Two Subtopics:

A. Web Security Threats
B. Web Traffic Security Approach
Web Security Considerations

• The WEB is very visible.
• Complex software hide many security flaws.
• Web servers are easy to configure and manage.
• Users are not aware of the risks.
a) Web Security Threats

- **Classification 1**: Threat Groups:
  - Active Attacks
  - Passive Attacks
- **Passive attacks** include eavesdropping on network traffic between browser and server and gaining access to information on a Web site that is supposed to be restricted.
- **Active attacks** include impersonating another user, altering messages in transit between client and server, and altering information on a Web site.
- Classification 1 already studied.
- **Classification 2**: Threat Location:
  - Web server,
  - Web browser, and
  - Network traffic between browser and server.
- Classification 2 we will study in this chapter.
- See next table providing summery of types of security threats.
<table>
<thead>
<tr>
<th>Threats</th>
<th>Consequences</th>
<th>Countermeasures</th>
</tr>
</thead>
</table>
| **Integrity** | • Loss of information  
• Compromise of machine  
• Vulnerability to all other threats | Cryptographic checksums |
| • Modification of user data  
• Trojan horse browser  
• Modification of memory  
• Modification of message traffic in transit | |
| **Confidentiality** | • Loss of information  
• Loss of privacy | Encryption, Web proxies |
| • Eavesdropping on the net  
• Theft of info from server  
• Theft of data from client  
• Info about network configuration  
• Info about which client talks to server | |
| **Denial of Service** | • Disruptive  
• Annoying  
• Prevent user from getting work done | Difficult to prevent |
| • Killing of user threads  
• Flooding machine with bogus requests  
• Filling up disk or memory  
• Isolating machine by DNS attacks | |
| **Authentication** | • Misrepresentation of user  
• Belief that false information is valid | Cryptographic techniques |
| • Impersonation of legitimate users  
• Data forgery | |
b) Web Traffic Security Approach:

- Security can be provided in **three ways**:
  - At Network Layer
  - At Transport Layer
  - At Application Layer

- **At Network Layer**:
  - Use IP Security
  - Advantage:
    - Transparent to users.
    - Include filtering capability (only selected traffic incur overhead of IPSec Processing)
Security facilities in the TCP/IP protocol stack

(a) Network Level
(b) Transport Level
(c) Application Level
• **Transport Layer:**
  - Implement security just above TCP.
  - Example: Secure Socket Layer (SSL) following Transport Layer Security.
  - Two choices of implementation:
    - SSL
    - SSL and TLS.
  - Example: Netscape and Microsoft Explorer browser comes equipped with SSL.

• **Application Layer:**
  - Advantage: Service can be tailored to specific need of given application.
  - Example: SET Secure Electronic Transactions.
SSL (Secure Socket Layer) & TLS (Transport Layer Security)
SSL and TLS

- SSL was originated by Netscape.
- TLS working group was formed within IETF.
- First version of TLS can be viewed as an SSLv3.1 which is very close to SSLv3 and backward compatible.
- Topic List:
  A. SSL Architecture
     a) SSL Record Protocol
     b) Change Cipher Spec Protocol
     c) Alert Protocol
     d) Handshake Protocol
  B. Cryptographic Computation
     a) Master Secret Creation
     b) Generation of Cryptographic Parameters
SSL Architecture

- SSL is designed to make use of TCP to provide a reliable end-to-end secure service.
- SSL is two layers of protocols.
- Components used to manage SSL:
  a) SSL Record Protocol
  b) Change Cipher Spec Protocol
  c) Alert Protocol
  d) Handshake Protocol
- Two Concepts:
  - Connections:
    - Each connection has Peer to peer relationship.
    - Each connection is associated to one session.
  - Sessions:
    - Association between client and server.
    - Sessions are created by Handshaking Protocol.
    - Sessions define set of cryptographic security parameter which is shared by connection.
    - Sessions avoid expensive negotiation of new security parameter.
- Lets see session and connection parameters:
A. SSL Architecture

![SSL Protocol Stack Diagram]

**Figure 7.2** SSL Protocol Stack
Connection Parameter

- **Server & Client Random**: Byte sequences that are chosen by each connection.
- **Server write MAC secret**: The secret key used in MAC operations on data sent by the server.
- **Client write MAC secret**: The secret key used in MAC operations on data sent by the client.
- **Server write key**: The secret encryption key for data encrypted by the server and decrypted by the client.
- **Client write key**: The symmetric encryption key for data encrypted by the client and decrypted by the server.
- **Initialization vectors**: When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key.
- **Sequence numbers**: Each party maintains separate sequence numbers for transmitted and received messages for each connection.
Session Parameter

- **Session identifier**: An arbitrary byte sequence chosen by the server to identify an active or resumable session state.
- **Peer certificate**: An X509.v3 certificate of the peer. This element of the state may be null.
- **Compression method**: The algorithm used to compress data prior to encryption.
- **Cipher spec**: Specifies the bulk data encryption algorithm (such as null, AES, etc.) and a hash algorithm (such as MD5 or SHA-1) used for MAC calculation. It also defines cryptographic attributes such as the hash_size.
- **Master secret**: 48-byte secret shared between the client and server.
- **Is resumable**: A flag indicating whether the session can be used to initiate new connections.
SSL Record Protocol Operation

Application Data

Fragment

Compress

Add MAC

Encrypt

Append SSL Record Header
SSL Record Protocol provides two services for SSL connections:
- Confidentiality
- Message Integrity

The Record Protocol **sender side** takes an application message to be transmitted,
- fragments the data into manageable blocks,
- optionally compresses the data,
- applies a MAC,
- encrypts,
- adds a header, and
- transmits the resulting unit in a TCP segment.

Received data **receiver side** are
- decrypted,
- verified,
- decompressed, and
- reassembled before being delivered to higher-level users.
• **MAC:**

\[
\text{hash(MAC\_write\_secret} \ || \ \text{pad\_2} \ || \ \text{hash(MAC\_write\_secret} \ || \ \text{pad\_1} \ || \ \text{seq\_num} \ || \ \text{SSLCompressed.type} \ || \\
\text{SSLCompressed.length} \ || \ \text{SSLCompressed.fragment}))
\]

• where

  - || = concatenation
  - MAC\_write\_secret = shared secret key
  - hash = cryptographic hash algorithm; either MD5 or SHA-1
  - pad\_1 = the byte 0x36 (0011 0110) repeated 48 times (384 bits) for MD5 and 40 times (320 bits) for SHA-1
  - pad\_2 = the byte 0x5C (0101 1100) repeated 48 times for MD5 and 40 times for SHA-1
  - seq\_num = the sequence number for this message
  - SSLCompressed.type = the higher-level protocol used to process this fragment
  - SSLCompressed.length = the length of the compressed fragment
  - SSLCompressed.fragment = the compressed fragment (if compression is not used, this is the plaintext fragment)
1. **Encrypted** using symmetric encryption shown in figure.

2. Final step: **Append header**. Has following fields:
   - **Content Type** (8 bits): The higher-layer protocol used to process the enclosed fragment.
   - **Major Version** (8 bits): Indicates major version of SSL in use. For SSLv3, the value is 3.
   - **Minor Version** (8 bits): Indicates minor version in use. For SSLv3, the value is 0.
   - **Compressed Length** (16 bits): The length in bytes of the plaintext fragment (or compressed fragment if compression is used). The maximum value is $2^{14} + 2048$.

<table>
<thead>
<tr>
<th>Block Cipher</th>
<th>Key Size</th>
<th>Stream Cipher</th>
<th>Algorithm</th>
<th>Key Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>128, 256</td>
<td>RC4-40</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>IDEA</td>
<td>128</td>
<td>RC4-128</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>RC2-40</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DES-40</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DES</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DES</td>
<td>168</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortezza</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SSL Record Format

<table>
<thead>
<tr>
<th>Content Type</th>
<th>Major Version</th>
<th>Minor Version</th>
<th>Compressed Length</th>
</tr>
</thead>
</table>

encrypted

Plaintext (optionally compressed)

MAC (0, 16, or 20 bytes)
### SSL Record Protocol Payload

**1 byte**

<table>
<thead>
<tr>
<th>1</th>
<th>3 bytes</th>
<th>0 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Length</td>
<td>Content</td>
</tr>
</tbody>
</table>

(a) Change Cipher Spec Protocol

(b) Alert Protocol

(c) Handshake Protocol

(d) Other Upper-Layer Protocol (e.g., HTTP)
Change Cipher Spec Protocol

• One of the three SSL Specific Protocols that use SSL Record Protocol.
• It is simplest.
• Consist of single msg (a) having single byte with value 1.
• Purpose: Cause pending state to be copied to current state, which updates cipher suite to be used on this connection.
Alert Protocol

- Use: convey SSL-related alerts to the peer entity.
- Alert messages are compressed and encrypted.
- Each message in this protocol consists of two bytes (Figure 5.5b).
- **First Byte:**
  - Takes the value warning (1) or fatal (2) to convey the severity of the message.
  - If the level is fatal, SSL immediately terminates the connection.
  - Other connections on the same session may continue, but no new connections on this session may be established.
- **Second Byte:**
  - The second byte contains a code that indicates the specific alert.
• **Always Fatal Alerts:**
  - `unexpected_message`: An inappropriate message was received.
  - `bad_record_mac`: An incorrect MAC was received.
  - `decompression_failure`: The decompression function received improper input (e.g., unable to decompress or decompress to greater than maximum allowable length).
  - `handshake_failure`: Sender was unable to negotiate an acceptable set of security parameters given the options available.
  - `illegal_parameter`: A field in a handshake message was out of range or inconsistent with other fields.
• **Other Alerts:**
  - **close_notify**: Notifies the recipient that the sender will not send any more messages on this connection. Required before closing connection.
  - **no_certificate**: May be sent in response to a certificate request if no appropriate certificate is available.
  - **bad_certificate**: A received certificate was corrupt (e.g., contained a signature that did not verify).
  - **unsupported_certificate**: The type of the received certificate is not supported.
  - **certificate_revoked**: A certificate has been revoked by its signer.
  - **certificate_expired**: A certificate has expired.
  - **certificate_unknown**: Some other unspecified issue arose in processing the certificate, rendering it unacceptable.
Handshake Protocol

- The most **complex** part of SSL.
- Allows the server and client to **authenticate** each other.
- **Negotiate** encryption, MAC algorithm and cryptographic keys.
- Used before **any** application data are transmitted.
- Consist of **series** of msg exchanged by client and server having **format** Fig. C.
- **Has 3 fields:**
  - **Type** (1 byte): Indicates one of 10 messages. Next slide lists the defined message types.
  - **Length** (3 bytes): The length of the message in bytes.
  - **Content** (bytes): The parameters associated with this message; these are listed in next table.
- The process is divided into 4 **phase**:
  - Phase 1: Establish Security Capabilities
  - Phase 2: Server Authentication & Key Exchange
  - Phase 3: Client Authentication & Key Exchange
  - Phase 4: Finish
<table>
<thead>
<tr>
<th>Message Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hello_request</strong></td>
<td>null</td>
</tr>
<tr>
<td><strong>client_hello</strong></td>
<td>version, random, session id, cipher suite, compression method</td>
</tr>
<tr>
<td><strong>server_hello</strong></td>
<td>version, random, session id, cipher suite, compression method</td>
</tr>
<tr>
<td><strong>certificate</strong></td>
<td>chain of X.509v3 certificates</td>
</tr>
<tr>
<td><strong>server_key_exchange</strong></td>
<td>parameters, signature</td>
</tr>
<tr>
<td><strong>certificate_request</strong></td>
<td>type, authorities</td>
</tr>
<tr>
<td><strong>server_done</strong></td>
<td>null</td>
</tr>
<tr>
<td><strong>certificate_verify</strong></td>
<td>signature</td>
</tr>
<tr>
<td><strong>client_key_exchange</strong></td>
<td>parameters, signature</td>
</tr>
<tr>
<td><strong>finished</strong></td>
<td>hash value</td>
</tr>
</tbody>
</table>
Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers.

Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.

Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.

Change cipher suite and finish handshake protocol.

Note: Shaded transfers are optional or situation-dependent messages that are not always sent.
Phase 1: Establish Security Capabilities

- Initiate logical connection.
- Establish security capabilities.
- Exchange is initiated by client with `client_hello` message.
- Has following parameters:
  - Version
  - Random (32bit timestamp, 28 bytes as nonce to prevent replay attack)
  - Session ID
  - Cipher-Suite (This is a list that contains the combinations of cryptographic algorithms supported by the client, in decreasing order of preference. Each cipher suite defines both a key exchange algorithm and a CipherSpec.)
  - Compressed Method (client supported)
- Then client waits for the `server_hello` message, where following conventions apply:
  - Version field contains the lower of the versions suggested by the client and the highest supported by the server.
  - The Random field is generated by the server and is independent of the client’s Random field.
  - If the SessionID field of the client was nonzero, the same value is used by the server; otherwise the server’s SessionID field contains the value for a new session.
  - The CipherSuite field contains the single cipher suite agreed by client and server both.
  - Compression field contains the compression method.
• **CipherSuite** Key Exchange methods:
  - **RSA**: Uses RSA public key to encrypt secret key.
  - **Fixed Diffie-Hellman**: This method results in a fixed secret key between two peers based on the Diffie-Hellman calculation using the fixed public keys.
  - **Ephemeral Diffie-Hellman**: This technique is used to create ephemeral (temporary, one-time) secret keys.
  - **Anonymous Diffie-Hellman**: Each side sends its public Diffie-Hellman parameters to the other with no authentication.
  - **Fortezza**: The technique defined for the Fortezza scheme.
- **Cipher Algorithm**: Any of the algorithms mentioned earlier: RC4, RC2, DES, 3DES, DES40, IDEA, or Fortezza
- **MAC Algorithm**: MD5 or SHA-1
- **Cipher Type**: Stream or Block
- **Is Exportable**: True or False
- **Hash Size**: 0, 16 (for MD5), or 20 (for SHA-1) bytes
- **Key Material**: A sequence of bytes that contain data used in generating the write keys
- **IV Size**: The size of the Initialization Value for Cipher Block Chaining (CBC) encryption
Phase 2: Server Authentication & Key Exchange

- **Msg** contains one or a chain of X.509 certificates.
- The **certificate message** is required for any agreed-on key exchange method except anonymous Diffie-Hellman.
- Next, a **server_key_exchange** message may be sent if it is required.
- It is not required in two instances:
  - (1) The server has sent a certificate with fixed Diffie-Hellman parameters or
  - (2) a RSA key exchange is to be used.
- The **server_key_exchange** message is needed for the following:
  - Anonymous Diffie-Hellman
  - Ephemeral Diffie-Hellman
  - RSA key exchange
  - Fortezza
- **Hash** is defined as
  \[ \text{hash(ClientHello.random || ServerHello.random || ServerParams)} \]
Next, **certificate_request message**, a nonanonymuous server (server not using anonymous Diffie-Hellman) can request a certificate from the client.

They includes two parameters:
- certificate_type and
- certificateAuthorities.

**Certificate_Type** indicate public key algorithm and it uses:

1. RSA, signature only
2. DSS, signature only
3. RSA for fixed Diffie-Hellman; in this case the signature is used only for authentication, by sending a certificate signed with RSA
4. DSS for fixed Diffie-Hellman; again, used only for authentication
5. RSA for ephemeral Diffie-Hellman
6. DSS for ephemeral Diffie-Hellman
7. Fortezza

One that is always required, is the **server_done message**, indicate the end of the server hello and has no parameters.

After sending this message, the server will wait for a client response.
Phase 3: Client Authentication & Key Exchange

- Upon receipt of the server_done message, the client should verify that the server provided a valid certificate (if required) and check that the server_hello parameters are acceptable.
- If all is satisfactory, the client sends one or more messages back to the server.
- Client begins this phase by sending a certificate message.
- If no suitable certificate is available, the client sends a no_certificate alert instead.
- Next is the client_key_exchange message, which must be sent in this phase.
- The content of the message depends on the type of key exchange:
  - RSA
  - Ephemeral or Anonymous Diffie-Hellman
  - Fixed Diffie-Hellman
  - Fortezza
- Finally, client may send a certificate_verify message to provide explicit verification of a client certificate.
- This message signs a hash code based on the preceding messages, defined as ...
CertificateVerify.signature.md5_hash=
MD5(master_secret || pad_2 || MD5(handshake_messages ||
master_secret || pad_1));
CertificateVerify.signature.sha_hash=
SHA(master_secret || pad_2 || SHA(handshake_messages ||
master_secret || pad_1));

Where

- pad_1 and pad_2 are defined in MAC.
- handshake_messages refers to all Handshake Protocol messages sent or received starting at client_hello.
- master_secret is the calculated secret.
- Purpose: is to verify the client’s ownership of the private key for the client certificate.
- Even if someone is misusing the client’s certificate, he or she would be unable to send this message.
Phase 4: Finish

- This phase completes the setting up of a secure connection.
- Client sends a `change_cipher_spec` message and copies the pending CipherSpec into the current CipherSpec.
- **Note:** This msg is not part of the Handshake Protocol but Change Cipher Spec Protocol.
- The client then immediately sends the `finished` message which verifies that the key exchange and authentication processes were successful. It has 2 hash values given below.
- In response to these two messages, the server sends its own `change_cipher_spec` message, transfers the pending to the current CipherSpec, and sends its `finished` message.
- At this point, the handshake is complete and the client and server may begin to exchange application-layer data.

```
MD5(master_secret || pad2 || MD5(handshake_messages || Sender || master_secret || pad1))
SHA(master_secret || pad2 || SHA(handshake_messages || Sender || master_secret || pad1))
```
Cryptographic Computation:

- Includes following two:
  - Master Secret Creation
  - Generation of Cryptographic Parameters
Master Secret Creation:

- Shared master secret is a one-time 48-byte value (384 bits) generated for this session by means of secure key exchange.
- **Two stages:**
  - pre_master_secret
  - master_secret
- For pre_master_secret, following two possibilities:
  - RSA
  - Diffie-Hellman
- Both sides now compute the master_secret as given below.
- Where ClientHello.random and ServerHello.random are the two nonce values exchanged in the initial hello messages.

```
master_secret = MD5(pre_master_secret || SHA('A' || pre_master_secret || ClientHello.random || ServerHello.random)) ||
MD5(pre_master_secret || SHA('BB' || pre_master_secret || ClientHello.random || ServerHello.random)) ||
MD5(pre_master_secret || SHA('CCC' || pre_master_secret || ClientHello.random || ServerHello.random))
```
Generation of Cryptographic Message

- CipherSpecs require a client write MAC secret, a server write MAC secret, a client write key, a server write key, a client write IV, and a server write IV, which are generated from the master secret in that order.
- These parameters are generated from the master secret by hashing the master secret into a sequence of secure bytes of sufficient length for all needed parameters.
- Generation of `key_block` goes on until enough output has been generated.
- The result of this algorithmic structure is a pseudorandom function.

\[
\text{key\_block} = \text{MD5(master\_secret || SHA('A' || master\_secret || ServerHello.random || ClientHello.random)) || MD5(master\_secret || SHA('BB' || master\_secret || ServerHello.random || ClientHello.random)) || MD5(master\_secret || SHA('CCC' || master\_secret || ServerHello.random || ClientHello.random)) || ...}
\]
Transport Layer Security

• The same record format as the SSL record format.
• Defined in RFC 2246.
• Similar to SSLv3.
• Differences in the:
  a) version number
  b) message authentication code
  c) pseudorandom function
  d) alert codes
  e) cipher suites
  f) client certificate types
  g) certificate_verify and finished message
  h) cryptographic computations
  i) padding
a) Version Number

• The TLS Record Format is the same as that of the SSL Record Format (Figure 5.4).

• For the current version of TLS, the major version is 3 and the minor version is 3 (which is 3 and 0 for SSL).
b) Message Authentication Code

- There are two differences between the SSLv3 and TLS MAC schemes:
  - the actual algorithm and
  - the scope of the MAC calculation.
- TLS makes use of the HMAC algorithm defined in RFC 2104.

\[
\text{HMAC}_K(M) = H[(K^+ \oplus \text{opad}) \| H[(K^+ \oplus \text{ipad}) \| M]]
\]
Where,
- \( H \) = embedded hash function (for TLS, either MD5 or SHA-1)
- \( M \) = message input to HMAC
- \( K^+ \) = secret key padded with zeros on the left so that the result is equal to the block length of the hash code (for MD5 and SHA-1, block length = 512 bits)
- \( \text{ipad} = 00110110 \) (36 in hexadecimal) repeated 64 times (512 bits)
- \( \text{opad} = 01011100 \) (5C in hexadecimal) repeated 64 times (512 bits)

SSLv3 uses the same algorithm.

Only difference is padding bytes are concatenated with the secret key rather than being XORed with the secret key.

The MAC calculation covers all of the fields covered by the SSLv3 calculation, plus the field TLSCompressed.version, which is the version of the protocol being employed.

\[
\text{MAC(MAC\_write\_secret,seq\_num || TLSCompressed\_type || TLSCompressed\_version || TLSCompressed\_length || TLSCompressed\_fragment)}
\]
c) Pseudorandom Function

- Need? TLS makes use of a pseudorandom function referred to as PRF to expand secrets into blocks of data for purposes of key generation or validation.
- Objective: is to make use of a relatively small shared secret value but to generate longer blocks of data in a way that is secure from the kinds of attacks made on hash functions and MACs.
- PRF is based on the data expansion function.

\[
P_{\text{hash}}(\text{secret, seed}) = \text{HMAC}\_\text{hash}(\text{secret, } A(1) \ || \ \text{seed}) \ || \\
\text{HMAC}\_\text{hash}(\text{secret, } A(2) \ || \ \text{seed}) \ || \\
\text{HMAC}\_\text{hash}(\text{secret, } A(3) \ || \ \text{seed}) \ || \ldots
\]

where \( A() \) is defined as

\[
A(0) = \text{seed} \\
A(i) = \text{HMAC}\_\text{hash}(\text{secret, } A(i - 1))
\]
Figure 5.7  TLS Function \( p_{\text{hash}}(\text{secret}, \text{seed}) \)
• The data expansion function makes use of the HMAC algorithm with either MD5 or SHA-1 as the underlying hash function.

• \text{P\_hash} can be iterated as many times as necessary to produce the required quantity of data.

• To make PRF as secure as possible, it uses two hash algorithms.

• PRF is defined as given below.

• PRF takes as input a secret value, an identifying label, and a seed value and produces an output of arbitrary length.

\[
\text{PRF}(\text{secret}, \text{label}, \text{seed}) = \text{P\_hash}(S1, \text{label} \parallel \text{seed})
\]
d) Alert Codes

- **TLS supports** all of the alert codes defined in SSLv3 with the **exception** of **no_certificate**.
- Following are always fatal:
  - **record_overflow**: Record received whose length exceeds bytes, or the ciphertext decrypted to a length of greater than bytes.
  - **unknown_ca**: certificate was not accepted because the CA certificate could not be located or could not be matched with a known, trusted CA.
  - **access_denied**: valid certificate was received, but sender decided not to proceed with the negotiation.
  - **decode_error**: a field was out of its specified range or the length of the message was incorrect.
  - **protocol_version**: client attempted to negotiate is recognized but not supported.
- **insufficient_security**: Returned instead of **handshake_failure** when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client.
- **unsupported_extension**: Sent by clients that receive an extended server hello containing an extension not in the corresponding client hello.
- **internal_error**: An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue.
- **decrypt_error**: A handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a finished message.
- **user_canceled**: This handshake is being canceled for some reason unrelated to a protocol failure.
- **no_renegotiation**: this alert indicates that the sender is not able to renegotiate. This message is always a warning.
e) Cipher Suites

- **Key Exchange**: TLS supports all of the key exchange techniques of SSLv3 with the exception of Fortezza.

- **Symmetric Encryption Algorithms**: TLS includes all of the symmetric encryption algorithms found in SSLv3, with the exception of Fortezza.
f) Client Certificate Types:

- TLS defines the following certificate types to be requested in a `certificate_request` message:
  - `rsa_sign`,
  - `dss_sign`,
  - `rsa_fixed_dh`,
  - `dss_fixed_dh`.
- SSLv3 includes
  - `rsa_ephemeral_dh`,
  - `dss_ephemeral_dh`,
  - `fortezza_kea`
- TLS does not include the Fortezza scheme.
g) Certificate Verify & Finished Message

- In the TLS `certificate_verify` message, the MD5 and SHA-1 hashes are calculated only over `handshake_messages`.
- Master secret and pads are extra fields which add no additional security.
- For TLS, we have

\[
\operatorname{PRF}(\text{master\_secret}, \text{finished\_label}, \text{MD5(handshake\_messages)}) || \text{SHA-1(handshake\_messages)}
\]
h) Cryptographic Computation

- The `pre_master_secret` for TLS is calculated in the same way as in SSLv3.
- TLS calculation is different from that of SSLv3 and is defined as

\[
\text{master_secret} = \text{PRF}(\text{pre_master_secret}, \text{"master secret"}, \text{ClientHello.random} || \text{ServerHello.random})
\]
• The algorithm is performed until 48 bytes of pseudorandom output are produced.
• The calculation of the key block material (MAC secret keys, session encryption keys, and IVs) is defined as given below until enough output has been generated.
• The key_block is a function of the master_secret and the client and server random numbers, but for TLS, the actual algorithm is different.

```
key_block = PRF(master_secret, "key expansion",
                 SecurityParameters.server_random||
                 SecurityParameters.client_random)
```
i) Padding

• In SSL, the padding added prior to encryption, because the total size of the data to be encrypted is a multiple of the cipher's block length.
• In TLS, the padding can be any amount that results in a total that is a multiple of the cipher's block length, up to a maximum of 255 bytes.
• MAC plus padding.length byte is 79 bytes long, then the padding length (in bytes) can be 1, 9, 17, and so on, up to 249
• Used to frustrate attacks based on an analysis of the lengths of exchanged messages.
Secure Electronic Transactions

- An open encryption and security specification.
- Protect credit card transaction on the Internet.
- Companies involved:
  - MasterCard, Visa, IBM, Microsoft, Netscape, RSA, Terisa and Verisign
- Not a payment system.
- Set of security protocols and formats.
- SET is a single application with a single set of requirements, whereas IPSec and SSL/TLS are intended to support a range of applications.
SET Services

Three Services:
• Provides a secure communication channel in a transaction.
• Provides trust by the use of X.509v3 digital certificates.
• Ensures privacy.
SET Overview

• Requirements:
  - Provide confidentiality of payment and ordering information.
  - Ensure the integrity of all transmitted data.
  - Provide authentication that a cardholder is a legitimate user of a credit card account.
  - Provide authentication that a merchant can accept credit card transactions through its relationship with a financial institution.
  - Ensure the use of the best security practices and system design techniques to protect all legitimate parties in an electronic commerce transaction.
  - Creates a protocol that neither depends on transport security mechanisms nor prevents their use.
  - Facilitate and encourage interoperability among software and network providers.

• Key Features of SET:
  - Confidentiality of information
  - Integrity of data
  - Cardholder account authentication
  - Merchant authentication
SET Participants

- **Cardholder**
- **Merchant**
- **Issuer** (financial institute, bank)
- **Acquirer** (financial institute dealing with merchant)
- **Payment gateway** (The merchant exchanges SET messages with the payment gateway over the Internet, while the payment gateway has some direct or network connection to the acquirer's financial processing system.)
- **Certification authority** (entity that is trusted to issue X'509v3 public-key certificates for cardholders, merchants, and payment gateways.)
SET Participants

Cardholder

Certificate Authority

Issuer

Payment Network

Merchant

Internet

Acquirer

Payment Gateway
Sequence of events for transactions

1. The customer opens an account.
2. The customer receives a certificate.
3. Merchants have their own certificates.
4. The customer places an order.
5. The merchant is verified.
6. The order and payment are sent.
7. The merchant request payment authorization.
8. The merchant confirm the order.
9. The merchant provides the goods or service.
10. The merchant requests payments.
Dual Signature

\[ DS = E_{KR_c} [H(H(PI) \parallel H(OI))] \]

- **PI** = Payment Information
- **OI** = Order Information
- **H** = Hash function (SHA-1)
- **ll** = Concatenation
- **PI MD** = PI message digest
- **OI MD** = OI message digest
- **POMD** = Payment Order message digest
- **E** = Encryption (RSA)
- **KR_c** = Customer's private signature key
The customer wants to send the order information (OI) to the merchant and the payment information (PI) to the bank.

The merchant does not need to know the customer's credit card number, and the bank does not need to know the details of the customer's order.

What if the customers send the merchant two messages—
- a signed OI and
- a signed PI

And Merchant replace other OI with actual OI?

It is not possible. Why? Merchant would then have to find another OI whose hash matches the existing OIMD. With SHA-1, this is deemed not to be feasible. Thus, the merchant cannot link another OI with this PI.

Suppose that the merchant is in possession of the dual signature (DS), the OI, and the message digest for the PI (PIMD), they can compute hash value as given below.

And suppose the bank is in possession of DS, PI, the message digest for OI (OIMD) they can compute hash value as given below.

If these two quantities are equal, then the bank has verified the signature.

In summary,
1. The merchant has received OI and verified the signature.
2. The bank has received PI and verified the signature.
3. The customer has linked the OI and PI and can prove the linkage.

\[ H(\text{PIMD} \ || \ H(\text{OI})) \text{ and } D_{KU_c}[\text{DS}] \]

\[ H(H(\text{PI}) \ || \ \text{OIMD}) \text{ and } D_{KU_c}[\text{DS}] \]
Payment Processing

• Table 7.3 lists the transaction types supported by SET.
• Out of this we look in some detail at the following transactions:
  - Purchase request
  - Payment authorization
  - Payment capture
<table>
<thead>
<tr>
<th>SET Transaction Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardholder registration</td>
</tr>
<tr>
<td>Cardholders must register with a CA before they can send SET messages to merchants.</td>
</tr>
<tr>
<td>Merchant registration</td>
</tr>
<tr>
<td>Merchants must register with a CA before they can exchange SET messages with customers and payment gateways.</td>
</tr>
<tr>
<td>Purchase request</td>
</tr>
<tr>
<td>Message from customer to merchant containing OI for merchant and PI for bank.</td>
</tr>
<tr>
<td>Payment authorization</td>
</tr>
<tr>
<td>Exchange between merchant and payment gateway to authorize a given amount for a purchase on a given credit card account.</td>
</tr>
<tr>
<td>Payment capture</td>
</tr>
<tr>
<td>Allows the merchant to request payment from the payment gateway.</td>
</tr>
<tr>
<td>Certificate inquiry and status</td>
</tr>
<tr>
<td>If the CA is unable to complete the processing of a certificate request quickly, it will send a reply to the cardholder or merchant indicating that the requester should check back later. The cardholder or merchant sends the Certificate Inquiry message to determine the status of the certificate request and to receive the certificate if the request has been approved.</td>
</tr>
<tr>
<td>Purchase inquiry</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Authorization reversal</td>
</tr>
<tr>
<td>Capture reversal</td>
</tr>
<tr>
<td>Credit</td>
</tr>
<tr>
<td>Credit reversal</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Payment gateway certificate request</td>
</tr>
<tr>
<td>Batch administration</td>
</tr>
<tr>
<td>Error message</td>
</tr>
</tbody>
</table>
a) Purchase Request

• First of all cardholder has completed browsing, selecting, and ordering without help of SET.
• The purchase request exchange consists of four messages:
  - Initiate Request,
  - Initiate Response,
  - Purchase Request, and
  - Purchase Response.
• Cardholder must have a copy of the
  - certificates of the merchant and
  - the payment gateway.
• The message includes the following:
  1. Purchase-related information.
  2. Order-related information.
  3. Cardholder Certificate.
Payment processing

Cardholder sends Purchase Request

PI = Payment Information
OI = Order Information
PIMD = PI message digest
OIMD = OI message digest
E = Encryption (RSA for asymmetric; DES for symmetric)
K_S = Temporary symmetric key
K_U_b = Bank’s public key-exchange key
b) Payment Authorization

- The merchant authorizes the transaction with the payment gateway.
- This authorization guarantees that the merchant will receive payment.
- Consists of two messages:
  - Authorization Request and
  - Authorization response.
- Authorization Request message to the payment gateway consisting of the following:
  1. Purchase-related information.
  3. Certificates.
Payment processing

Merchant Verifies Customer Purchase Request
Payment Capture

• To obtain payment, the merchant engages the payment gateway in a payment capture transaction.

• **Payment Capture:**
  - Capture Request
  - Capture Response

• For the **Capture Request** message, the merchant generates, signs, and encrypts a capture request block, which includes the payment amount and the transaction ID.
• The message also includes the encrypted capture token.
• When the payment gateway receives the capture request message, it decrypts and verifies the capture request block and decrypts and verifies the capture token block.
• It then checks for consistency between the capture request and capture token. It then creates a clearing request that is sent to the issuer.
• This request causes funds to be transferred to the merchant's account.
• The gateway then notifies the merchant of payment in a **Capture Response** message.
• The message includes a capture response block that the gateway signs and encrypts.
• The message also includes the gateway's signature key certificate.
• The merchant software stores the capture response to be used for reconciliation with payment received from the acquirer.
Chapter Ends Here

• Last 25% syllabus and 2 Chapters left.
• Need not more than 15 Days.
• Start preparing for the exams.
• 😊😊😊😊😊
• Now pressure on me is reducing and yrs increasing......... 😊😊😊😊😊