What is SSL?

- SSL – Secure Socket Layer
- it provides a secure transport connection between applications (e.g., a web server and a browser)
- SSL was developed by Netscape
- SSL version 3.0 has been implemented in many web browsers (e.g., Netscape Navigator and MS Internet Explorer) and web servers and widely used on the Internet
- SSL v3.0 was specified in an Internet Draft (1996)
- it evolved into RFC 2246 and was renamed to TLS v1.0 (Transport Layer Security)
- current version is TLS v1.1 (RFC 4346)
  - modifications to handle CBC attacks: explicit IV and bad_record_mac error message instead of decryption_failed
Secure Socket Layer (SSL)

SSL architecture

- **SSL Handshake Protocol**
  - negotiation of security algorithms and parameters
  - key exchange
  - server authentication and optionally client authentication

- **SSL Record Protocol**
  - fragmentation
  - compression
  - message authentication and integrity protection
  - encryption

- **SSL Alert Protocol**
  - error messages (fatal alerts and warnings)

- **SSL Change Cipher Spec Protocol**
  - a single message that indicates the end of the SSL handshake

SSL components
Sessions and connections

- an SSL session is an association between a client and a server
- sessions are stateful; the session state includes security algorithms and parameters
- a session may include multiple secure connections between the same client and server
- connections of the same session share the session state
- sessions are used to avoid expensive negotiation of new security parameters for each connection
- there may be multiple simultaneous sessions between the same two parties, but this feature is not used in practice

Session state

- session identifier
  - arbitrary byte sequence chosen by the server to identify the session
- peer certificate
  - X509 certificate of the peer
  - may be null
- compression method
- cipher spec
  - bulk data encryption algorithm (e.g., null, DES, 3DES, …)
  - MAC algorithm (e.g., MD5, SHA-1)
  - cryptographic attributes (e.g., hash size, IV size, …)
- master secret
  - 48-byte secret shared between the client and the server
- is resumable
  - a flag indicating whether the session can be used to initiate new connections
- connection states
  - see next slide…
### Connection state

- server and client random
  - random byte sequences chosen by the server and the client for every connection
- server write MAC secret
  - secret key used in MAC operations on data sent by the server
- client write MAC secret
  - secret key used in MAC operations on data sent by the client
- server write key
  - secret encryption key for data encrypted by the server
- client write key
  - secret encryption key for data encrypted by the client
- initialization vectors
  - an IV is maintained for each encryption key if CBC mode is used
  - initialized by the SSL Handshake Protocol
  - final ciphertext block from each record is used as IV with the following record
- sending and receiving sequence numbers
  - sequence numbers are 64 bits long
  - reset to zero after each Change Cipher Spec message

### State change

- operating state
  - currently used state
- pending state
  - state to be used
    - built using the current state
- operating state ← pending state
  - at the transmission and reception of a Change Cipher Spec message
SSL Record Protocol – processing overview

- fragmentation
- compression
- MAC computation
- padding
- encryption

→ SSL Record Protocol message:

<table>
<thead>
<tr>
<th>type</th>
<th>version</th>
<th>length</th>
</tr>
</thead>
</table>

application data  
(compressed fragment)

MAC

<table>
<thead>
<tr>
<th>padding</th>
<th>p.len</th>
</tr>
</thead>
</table>

Header

- type
  - the higher level protocol used to process the enclosed fragment
  - possible types:
    - change_cipher_spec
    - alert
    - handshake
    - application_data
- version
  - SSL version, currently 3.0
- length
  - length (in bytes) of the enclosed fragment or compressed fragment
  - max value is $2^{14} + 2048$
MAC

MAC = hash( MAC_wr_sec | pad_2 |
             hash( MAC_wr_sec | pad_1 | seq_num | type | length | frag ))

- similar to HMAC but the pads are concatenated
- supported hash functions:
  - MD5
  - SHA-1
- pad_1 is 0x36 repeated 48 times (MD5) or 40 times (SHA-1)
- pad_2 is 0x5C repeated 48 times (MD5) or 40 times (SHA-1)

Encryption

- supported algorithms
  - block ciphers (in CBC mode)
    • RC2_40
    • DES_40
    • DES_56
    • 3DES_168
    • IDEA_128
    • Fortezza_80
  - stream ciphers
    • RC4_40
    • RC4_128
- if a block cipher is used, than padding is applied
  - last byte of the padding is the padding length
SSL Alert Protocol

- each alert message consists of 2 fields (bytes)
- first field (byte): “warning” or “fatal”
- second field (byte):
  - fatal:
    - unexpected_message
    - bad_record_MAC
    - decryption_failure
    - ...
  - warning:
    - close_notify
    - no_certificate
    - bad_certificate
    - unsupported_certificate
    - ...
- in case of a fatal alert
  - connection is terminated
  - session ID is invalidated \(\rightarrow\) no new connection can be established within this session

SSL Handshake Protocol – overview

<table>
<thead>
<tr>
<th>client</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_hello</td>
<td>server_hello</td>
</tr>
<tr>
<td>certificate</td>
<td>server_key_exchange</td>
</tr>
<tr>
<td>certificate_request</td>
<td>server_hello_done</td>
</tr>
<tr>
<td>certificate</td>
<td>client_key_exchange</td>
</tr>
<tr>
<td>certificate_verify</td>
<td>change_cipher_spec</td>
</tr>
<tr>
<td>change_cipher_spec</td>
<td>finished</td>
</tr>
<tr>
<td>finished</td>
<td>change_cipher_spec</td>
</tr>
</tbody>
</table>

Phase 1: Negotiation of the session ID, key exchange algorithm, MAC algorithm, encryption algorithm, and exchange of initial random numbers

Phase 2: Server may send its certificate and key exchange message, and it may request the client to send a certificate. Server signals end of hello phase.

Phase 3: Client sends certificate if requested and may send an explicit certificate verification message. Client always sends its key exchange message.

Phase 4: Change cipher spec and finish handshake
Client hello message

- client_version
  - the highest version supported by the client
- client_random
  - current time (4 bytes) + pseudo random bytes (28 bytes)
- session_id
  - empty if the client wants to create a new session, or
  - the session ID of an old session within which the client wants to create the new connection
- cipher_suites
  - list of cryptographic options supported by the client ordered by preference
  - a cipher suite contains the specification of the
    - key exchange method, the encryption and the MAC algorithm
    - the algorithms implicitly specify the hash_size, IV_size, and key_material parameters (part of the Cipher Spec of the session state)
  - example: SSL_RSA_with_3DES_EDE_CBC_SHA
- compression_methods
  - list of compression methods supported by the client

Server hello message

- server_version
  - min( highest version supported by client, highest version supported by server )
- server_random
  - current time + random bytes
  - random bytes must be independent of the client random
- session_id
  - session ID chosen by the server
  - if the client wanted to resume an old session:
    - server checks if the session is resumable
    - if so, it responds with the session ID and the parties proceed to the finished messages
  - if the client wanted a new session
    - server generates a new session ID
- cipher_suite
  - single cipher suite selected by the server from the list given by the client
- compression_method
  - single compression method selected by the server
### Supported key exchange methods

- **RSA based (SSL_RSA_with...)**
  - the secret key (pre-master secret) is encrypted with the server’s public RSA key
  - the server’s public key is made available to the client during the exchange

- **fixed Diffie-Hellman (SSL_DH_RSA_with... or SSL_DH_DSS_with...)**
  - the server has fix DH parameters contained in a certificate signed by a CA
  - the client may have fix DH parameters certified by a CA or it may send an unauthenticated one-time DH public value in the client_key_exchange message

- **ephemeral Diffie-Hellman (SSL_DHE_RSA_with... or SSL_DHE_DSS_with...)**
  - both the server and the client generate one-time DH parameters
  - the server signs its DH parameters with its private RSA or DSS key
  - the client may authenticate itself (if requested by the server) by signing the hash of the handshake messages with its private RSA or DSS key

- **anonymous Diffie-Hellman (SSL_DH_anon_with...)**
  - both the server and the client generate one-time DH parameters
  - they send their parameters to the peer without authentication

- **Fortezza**
  - Fortezza proprietary key exchange scheme

### Server certificate and key exchange msgs

- **certificate**
  - required for every key exchange method except for anonymous DH
  - contains one or a chain of X.509 certificates (up to a known root CA)
  - may contain
    - public RSA key suitable for encryption, or
    - public RSA or DSS key suitable for signing only, or
    - fix DH parameters

- **server_key_exchange**
  - sent only if the certificate does not contain enough information to complete the key exchange (e.g., the certificate contains an RSA signing key only)
  - may contain
    - public RSA key (exponent and modulus), or
    - DH parameters \((p, g, \text{public DH value})\), or
    - Fortezza parameters
  - digitally signed
    - if DSS: SHA-1 hash of \((\text{client_random} \mid \text{server_random} \mid \text{server_params})\) is signed
    - if RSA: MD5 hash and SHA-1 hash of \((\text{client_random} \mid \text{server_random} \mid \text{server_params})\) are concatenated and encrypted with the private RSA key
Cert request and server hello done msgs

- **certificate_request**
  - sent if the client needs to authenticate itself
  - specifies which type of certificate is requested (rsa_sign, dss_sign, rsa_fixed_dh, dss_fixed_dh, …)

- **server_hello_done**
  - sent to indicate that the server is finished its part of the key exchange
  - after sending this message the server waits for client response
  - the client should verify that the server provided a valid certificate and the server parameters are acceptable

Client authentication and key exchange

- **certificate**
  - sent only if requested by the server
  - may contain
    - public RSA or DSS key suitable for signing only, or
    - fix DH parameters

- **client_key_exchange**
  - always sent (but it is empty if the key exchange method is fix DH)
  - may contain
    - RSA encrypted pre-master secret, or
    - client one-time public DH value, or
    - Fortezza key exchange parameters

- **certificate_verify**
  - sent only if the client sent a certificate
  - provides client authentication
  - contains signed hash of all the previous handshake messages
    - if DSS: SHA-1 hash is signed
    - if RSA: MD5 and SHA-1 hash is concatenated and encrypted with the private key
      MD5( master_secret | pad_2 | MD5( handshake_messages | master_secret | pad_1 ) )
      SHA( master_secret | pad_2 | SHA( handshake_messages | master_secret | pad_1 ) )
Finished messages

- **finished**
  - sent immediately after the change_cipher_spec message
  - used to authenticate all previous handshake messages
  - first message that uses the newly negotiated algorithms, keys, IVs, etc.
  - contains the MD5 and SHA-1 hash of all the previous handshake messages:
    \[
    \text{MD5}(\text{master_secret} \mid \text{pad}_2 \mid \text{MD5}(\text{handshake_messages} \mid \text{sender} \mid \text{master_secret} \mid \text{pad}_1)) \mid \\
    \text{SHA}(\text{master_secret} \mid \text{pad}_2 \mid \text{SHA}(\text{handshake_messages} \mid \text{sender} \mid \text{master_secret} \mid \text{pad}_1))
    \]
    where “sender” is a code that identifies that the sender is the client or the server (client: 0x434C4E54; server: 0x53525652)

Cryptographic computations

- **pre-master secret**
  - if key exchange is RSA based:
    - generated by the client
    - sent to the server encrypted with the server’s public RSA key
  - if key exchange is Diffie-Hellman based:
    - \(\text{pre_master_secret} = g^{xy} \mod p\)

- **master secret (48 bytes)**
  \[
  \text{master_secret} = \text{MD5}(\text{pre_master_sec} \mid \text{SHA}(\text{"A"} \mid \text{pre_master_sec} \mid \text{client_random} \mid \text{server_random})) \mid \\
  \text{MD5}(\text{pre_master_sec} \mid \text{SHA}(\text{"BB"} \mid \text{pre_master_sec} \mid \text{client_random} \mid \text{server_random})) \mid \\
  \text{MD5}(\text{pre_master_sec} \mid \text{SHA}(\text{"CCC"} \mid \text{pre_master_sec} \mid \text{client_random} \mid \text{server_random}))
  \]

- **keys, MAC secrets, IVs**
  \[
  \text{MD5}(\text{master_secret} \mid \text{SHA}(\text{"A"} \mid \text{master_secret} \mid \text{client_random} \mid \text{server_random})) \mid \\
  \text{MD5}(\text{master_secret} \mid \text{SHA}(\text{"BB"} \mid \text{master_secret} \mid \text{client_random} \mid \text{server_random})) \mid \\
  \text{MD5}(\text{master_secret} \mid \text{SHA}(\text{"CCC"} \mid \text{master_secret} \mid \text{client_random} \mid \text{server_random})) \mid \\
  \ldots
  \]

  - key block:
## Key exchange alternatives

- **RSA / no client authentication**
  - server sends its encryption capable RSA public key in `server_certificate`
  - `server_key_exchange` is not sent
  - client sends encrypted pre-master secret in `client_key_exchange`
  - `client_certificate` and `certificate_verify` are not sent
  or
  - server sends its RSA or DSS public signature key in `server_certificate`
  - server sends a temporary RSA public key in `server_key_exchange`
  - client sends encrypted pre-master secret in `client_key_exchange`
  - `client_certificate` and `certificate_verify` are not sent

## Key exchange alternatives cont’d

- **RSA / client is authenticated**
  - server sends its encryption capable RSA public key in `server_certificate`
  - `server_key_exchange` is not sent
  - client sends its RSA or DSS public signature key in `client_certificate`
  - client sends encrypted pre-master secret in `client_key_exchange`
  - client sends signature on all previous handshake messages in `certificate_verify`
  or
  - server sends its RSA or DSS public signature key in `server_certificate`
  - server sends a one-time RSA public key in `server_key_exchange`
  - client sends its RSA or DSS public signature key in `client_certificate`
  - client sends encrypted pre-master secret in `client_key_exchange`
  - client sends signature on all previous handshake messages in `certificate_verify`
### Key exchange alternatives cont’d

- **fix DH / no client authentication**
  - server sends its fix DH parameters in `server_certificate`
  - `server_key_exchange` is not sent
  - client sends its one-time DH public value in `client_key_exchange`
  - `client_certificate` and `certificate_verify` are not sent

- **fix DH / client is authenticated**
  - server sends its fix DH parameters in `server_certificate`
  - `server_key_exchange` is not sent
  - client sends its fix DH parameters in `client_certificate`
  - `client_key_exchange` is sent but empty
  - `certificate_verify` is not sent

---

### Key exchange alternatives cont’d

- **ephemeral DH / no client authentication**
  - server sends its RSA or DSS public signature key in `server_certificate`
  - server sends signed one-time DH parameters in `server_key_exchange`
  - client sends one-time DH public value in `client_key_exchange`
  - `client_certificate` and `certificate_verify` are not sent

- **ephemeral DH / client is authenticated**
  - server sends its RSA or DSS public signature key in `server_certificate`
  - server sends signed one-time DH parameters in `server_key_exchange`
  - client sends its RSA or DSS public signature key in `client_certificate`
  - client sends one-time DH public value in `client_key_exchange`
  - client sends signature on all previous handshake messages in `certificate_verify`
**Key exchange alternatives cont’d**

- anonymous DH / no client authentication
  - server_certificate is not sent
  - server sends (unsigned) one-time DH parameters in server_key_exchange
  - client sends one-time DH public value in client_key_exchange
  - client_certificate and certificate_verify are not sent

- anonymous DH / client is authenticated
  - not allowed

**Eavesdropping**

+ all application data is encrypted with a short term connection key
+ short term key is derived from per-connection salts (client and server randoms) and a strong shared secret (master secret) by hashing (one-way operation)
  + even if connection keys are compromised the master secret remains intact
+ different keys are used in each connection and in each direction of the connection
+ supported encryption algorithms are strong
**Traffic analysis**

- SSL doesn’t attempt to protect against traffic analysis
  - padding length is not random
  - no padding if a stream cipher is used (this is the default option)

- if SSL is used to protect HTTP traffic, then an attacker
  - can learn the length of a requested URL
  - can learn the length of the HTML data returned
  - could find out which URL was requested with high probability

**Replay attacks**

+ SSL protects against replay attacks by including an implicit sequence number in the MAC computation
  + prevents re-order and deletion of messages
+ sequence numbers are 64 bit long
  + practically never wraps around
Message authentication

+/- SSL uses a HMAC-like MAC
  - it actually uses an obsolete version of HMAC
  + HMAC is provably secure
  + MAC secret is 128 bits long
  + different MAC secrets are used in different directions and connections
  - the MAC doesn’t involve the version number (part of the message)
    - if the version number is ever used, then it should be covered by the MAC
    - if the version number is never used, then it should not be sent

CBC encryption with padding in SSL/TLS

- SSL padding
  - last byte is the length n of the padding (not including the last byte)
  - all padding bytes have the value n
  - examples for correct padding: 0, 11, 222, 3333, …
- verification of SSL padding:
  - if the last byte is n, then verify if the last n+1 bytes are all n
Padding oracle attack (reminder)

- send a random message to a TLS server
- the server will drop the message with overwhelming probability
  - either the padding is incorrect (the server responds with a DECRYPTION_FAILED alert)
  - or the MAC is incorrect with very high probability (the server responds with BAD_RECORD_MAC alert)
- if the response is BAD_RECORD_MAC, then the padding was correct → we get 1 bit of information!
- such an oracle can be used to decrypt any encrypted message (see slides on block encryption modes)
- problems in practice
  - alert messages are encrypted → BAD_RECORD_MAC and DECRYPTION_FAILED cannot be distinguished
    - measure timing between oracle call and oracle response
  - BAD_RECORD_MAC takes more time than DECRYPTION_FAILED
  - BAD_RECORD_MAC and DECRYPTION_FAILED are fatal errors → connection is closed after one oracle call
    - a password can still be broken if it is sent periodically to a server using TLS (a different session (and key) is used each time the password is sent, but the password is always the same)

Example: IMAP over TLS

- Outlook Express checks for new mail on the server periodically (every 5 minutes)
- each time the same password is sent for every folder
  XXXX LOGIN "username" "password"<0D><0A>
- it is possible to uncover the password using the attack as follows:
Cipher suite rollback attack

- In SSL 2.0, an attacker could force the use of an export-weakened encryption algorithm by modifying the list of supported cipher suites in the hello messages.
- This is prevented in SSL 3.0 by authenticating all handshake messages with the master secret (in the finished messages).
- The master secret itself is authenticated by other means:
  - For the client:
    - Implicit authentication via the server certificate
      - Only the server could decrypt the RSA encrypted pre-master secret
      - Only the server could compute the pre-master secret from the client’s public DH value
    - Explicit authentication via the server_key_exchange message (if sent)
      - Ephemeral DH parameters are signed by the server
  - For the server:
    - Explicit authentication via the certificate_verify message (if sent)
      - Certificate_verify is signed by the client
      - It involves the master secret

Dropping the change_cipher_spec msg

- Authentication in the finished message does not protect the change_cipher_spec message (it is not part of the handshake protocol!)
- This may allow the following attack:
  - Assume that the negotiated cipher suite includes only message authentication (no encryption)

![Diagram of SSL handshake](attachment:image_url)
Dropping the change_cipher_spec msg

- if the negotiated cipher suite includes encryption, then the attacks doesn’t work
  - client sends encrypted finished message
  - server expects clear finished message
  - the attacker cannot decrypt the encrypted finished message

- simplest fix: require reception of change_cipher_spec before processing the finished message
  - this seems to be obvious, but…
  - even Netscape’s reference SSL implementation SSLRef 3.0b1 allows processing finished messages without checking if a change_cipher_spec has been received
  - SSLRef 3.0b3 contains the fix

- another fix: include the change_cipher_spec message in the computation of the finished message
  - this would require a more radical change in the SSL specification

---

Key-exchange algorithm rollback

<table>
<thead>
<tr>
<th>client</th>
<th>man-in-the-middle</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_hello: SSL_RSA,...</td>
<td></td>
<td>client_hello: SSL_DHE,...</td>
</tr>
<tr>
<td>server_hello: SSL_RSA,...</td>
<td></td>
<td>server_hello: SSL_DHE,...</td>
</tr>
<tr>
<td>certificate: server signing key</td>
<td></td>
<td>certificate: server signing key</td>
</tr>
<tr>
<td>server_key_exchange: p, g, g^y mod p, signature</td>
<td></td>
<td>server_key_exchange: p, g, g^y mod p, signature</td>
</tr>
<tr>
<td>client_key_exchange: sec^y mod p</td>
<td></td>
<td>client_key_exchange: g^y mod p</td>
</tr>
<tr>
<td>recover sec by computing g^y mod p (this is easy since p is prime)</td>
<td></td>
<td>compute sec' as (g^y)^e mod p</td>
</tr>
<tr>
<td>finished: { hash(msgs, sec'), macsec'(…)} sec'</td>
<td></td>
<td>finished: { hash(msgs, sec), macsec(…)} sec</td>
</tr>
</tbody>
</table>
Key-exchange algorithm rollback

- SSL authenticates only the server’s (RSA or DH) parameters in the server_key_exchange message
- it doesn’t authenticate the context (key exchange algorithm in use) in which those parameters should be interpreted
- this is not compliant with the Horton principle!

- a fix:
  - hash all messages exchanged before the server_key_exchange message
  - include the hash in the signature in server_key_exchange message

Version rollback attacks

- SSL 3.0 implementations may still support SSL 2.0
- an attacker may change the client_hello message so that it looks like an SSL 2.0 client_hello
- as a result the client and the server will run SSL 2.0
- SSL 2.0 has serious security flaws
  - among other things, there are no finished messages to authenticate the handshake
  - the version rollback attack will go undetected

- fortunately, SSL 3.0 can detect version rollback
  - pre-master secret generated on SSL 3.0 enabled clients:
    ```c
    struct{
        ProtocolVersion client_version; // latest version supported by the client
        opaque random[46]; // random bytes
    } PreMasterSecret;
    ```
  - an SSL 3.0 enabled server detects the version rollback attack, when it runs an SSL 2.0 handshake but receives a pre-master secret that includes version 3.0 as the latest version supported by the client
**MAC usage**

- while the SSL Record Protocol uses HMAC (an early version), the SSL Handshake Protocol uses ad-hoc MACs at several points
  - certificate_verify:
    \[
    \text{hash}(\text{master_secret} | \text{pad}_2 | \text{hash}(\text{handshake_messages} | \text{master_secret} | \text{pad}_1))
    \]
  - finished:
    \[
    \text{hash}(\text{master_secret} | \text{pad}_2 | \text{hash}(\text{handshake_messages} | \text{sender} | \text{master_secret} | \text{pad}_1))
    \]

- in addition, these ad-hoc MACs involve the master secret
- this is dangerous, and SSL should use HMAC consistently

---

**Analysis summary**

- **SSL Record Protocol**
  - good protection against passive eavesdropping and active attacks
    - should better protect against traffic analysis (e.g., apply random padding)
    - should use the latest version of HMAC
  
- **SSL Handshake Protocol**
  - some active attacks are foiled
    - cipher suite rollback
    - version rollback
  - other active attacks could still be possible depending on how an implementation interprets the SSL specification
    - dropping change_cipher_spec messages
    - key-exchange algorithm rollback
  - ad-hoc MAC constructions should be replaced with HMAC

- overall: SSL 3.0 was an extremely important step toward practical communication security for Internet applications
SSL vs. TLS

- version number
  - for TLS 1.1 the version number is 3.2
- cipher suites
  - TLS doesn’t support Fortezza key exchange and Fortezza encryption
- padding
  - variable length padding is allowed (max 255 padding bytes)
- MAC
  - TLS uses the latest version of HMAC
  - the MAC covers the version field of the record header too
- certificate_verify message
  - in SSL, the hash contains the master_secret
  - in TLS, the hash is computed only over the handshake messages
- more alert codes

New pseudorandom function (PRF)

- \( P_{\text{hash}}(\text{secret}, \text{seed}) = \text{HMAC}_\text{hash}(\text{secret}, A(1) | \text{seed}) | \text{HMAC}_\text{hash}(\text{secret}, A(2) | \text{seed}) | \text{HMAC}_\text{hash}(\text{secret}, A(3) | \text{seed}) | \ldots \)
  
  where
  
  \[
  A(0) = \text{seed} \\
  A(i) = \text{HMAC}_\text{hash}(\text{secret}, A(i-1))
  \]

- \( \text{PRF}(\text{secret}, \text{label}, \text{seed}) = P_{\text{MD5}}(\text{secret_left}, \text{label} | \text{seed}) \oplus P_{\text{SHA}}(\text{secret_right}, \text{label} | \text{seed}) \)
Secure Socket Layer (SSL)

Usage of the new PRF

- **finished message**
  \[
  \text{PRF}( \text{master_secret},
  \begin{array}{c}
  \text{"client finished"},
  \\
  \text{MD5(handshake_messages) | SHA(handshake_messages) }
  \end{array}
  )
  \]

- **cryptographic computations**
  - pre-master secret is calculated in the same way as in SSL
  - master secret:
    \[
    \text{PRF}( \text{pre_master_secret},
    \begin{array}{c}
    \text{"master secret"},
    \\
    \text{client_random | server_random }
    \end{array}
    )
    \]
  - key block:
    \[
    \text{PRF}( \text{master_secret},
    \begin{array}{c}
    \text{"key expansion"},
    \\
    \text{server_random | client_random }
    \end{array}
    )
    \]
<table>
<thead>
<tr>
<th>Further readings</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ The TLS protocol v1.1, available on-line as RFC 4346</td>
</tr>
</tbody>
</table>