IMPLEMENTATION QUESTION

GROUP 23

Soumyadeep Basu, Kumar Utkarsh, Rohit Haolader, Raushan Raj, Suryasen Singh

V Semester BTech, Department of Information Technology,

Indian Institute of Information Technology, Allahabad, India.

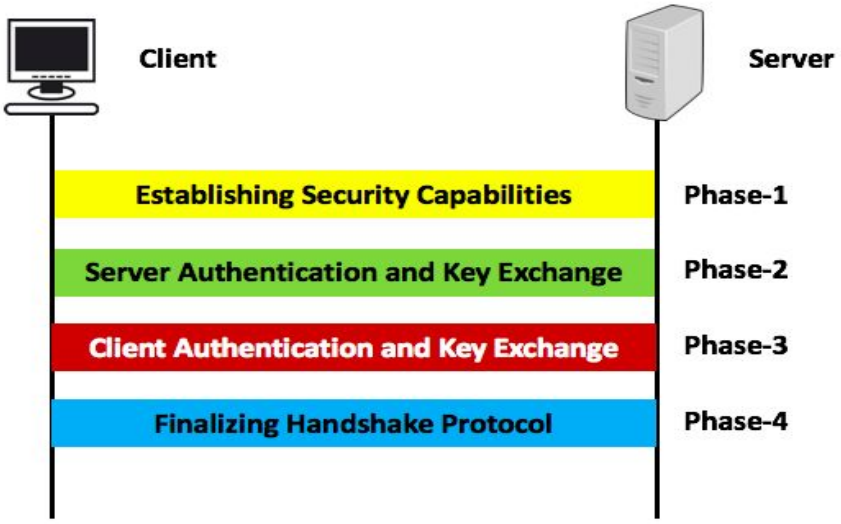
Phase 1 of the hand-shake protocol

SSL involves two important functions: (a) authentication of the communicating parties via the *SSL handshake protocol* and (b) encryption/decryption of data exchanged between the two parties during the session. The handshake protocol uses the public key infrastructure (PKI) and establishes a shared symmetric key between the parties to ensure confidentiality and integrity of the communicated data. The handshake involves three phases, with one or more messages exchanged between client and server:

Phase 1 starts with the client sending information regarding the algorithms and parameters that it can handle. This includes shared key exchange algorithm (e.g., RSA, Diffie-Hellman, etc.), bulk encryption algorithm to be used (e.g., AES, IDEA), and message digest algorithm (e.g., SHA1, SHA-256). Each of these algorithms have their own parameters (e.g., key length) that also must be communicated.

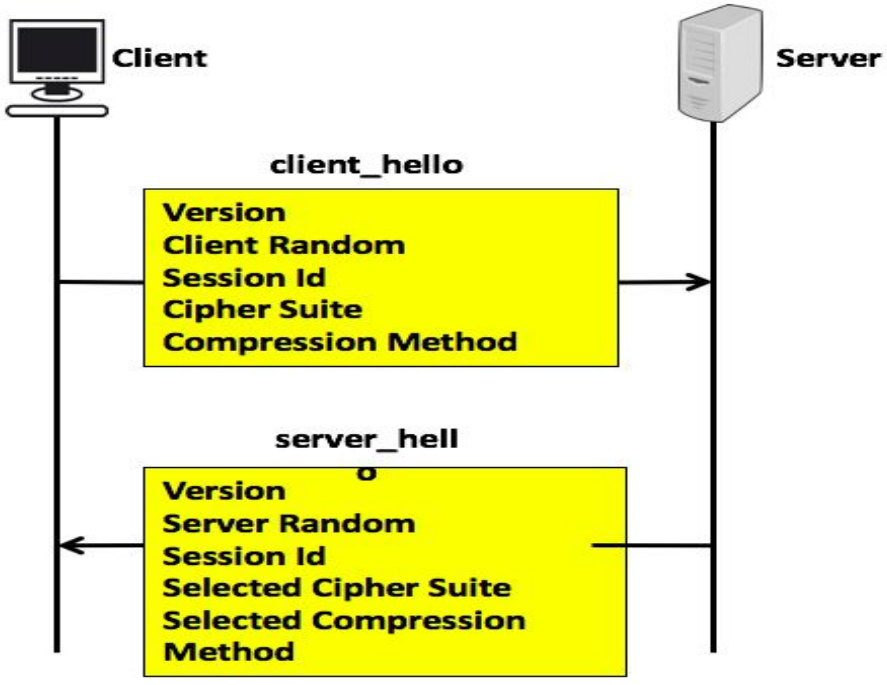
Phase 2 involves mutual authentication of client and server by exchanging their credentials. The server authentication is always based on a PKI digital certificate that a server must possess and supply to the client; however, the client rarely supplies its certificate. Instead, the client authentication is typically done following the handshake via a login/password supplied to the server.

Phase 3 involves the client sending the shared key to the server. This is encrypted with the server's public key so that only the addressed server is able to decipher it.



Visually, all these phases are showcased in this picture step by step and clearly. In this image, one client is trying to connect into a remote system/machine by using SSL (Secure Socket Layer). SSL is basically used over transport layers with HTTP protocol. for a few seconds forget about HTTPS and see how SSL works internally.

Phase-1: Establishing Securing Capabilities



In the first phase of establishing security capabilities is used to exchange security capabilities and started by client_hello message sent by the client to the server. It contains various parameters:

client_hello:

1. **version**:
2. **client random** : 32 bits timestamp + 28 bytes of random generated by client
3. **session Id**: variable session length, 0 mean new session, else client wants to update existing session.
4. **cipher Suite** : list of the course for decreasing order like keys, encryption methodology, etc..
5. **compression Method**: method which is used for compression etc.

server_hello

1. **version**: either send by client or server version if
2. **server random**: similar type of client session but independent of client.
3. **session Id**: if client id is 0 server put new session id which indicates new session else client id
4. **selected Cipher Suite**: selected suites by client
5. **selected compression method**: selected compression algorithm used during transfer.

ClientHello message contains:

• **version** = highest TLS version # the client can support

• **random** = 32-bit timestamp & 28 bytes generated by a secure random number generator – to serve as nonces to prevent replay attacks and are used during pre-master key exchange

• **session ID** = variable length session identifier – a non-zero value indicates that client wishes to update the parameters of an existing connection or to create a new connection on this session; a zero value indicates that client wishes to establish a new connection on a new session

• **CipherSuite** = list that contains cryptographic algorithms supported by the client, in decreasing order of preference

• **compression method** = list of compression methods that client can supports

ServerHello message contains:

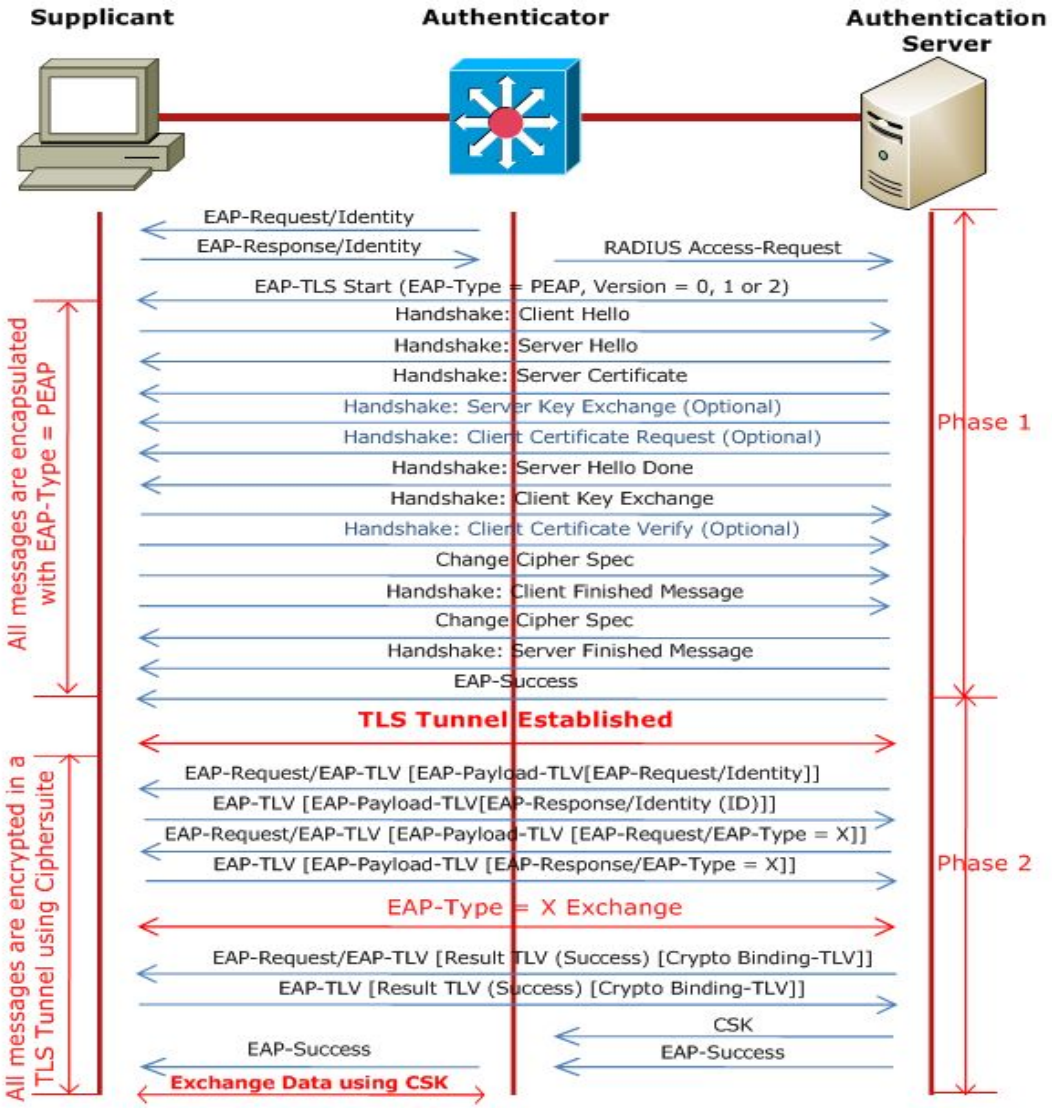
• **version** = the lower of two version numbers: the highest supported by client and highest supported by server

• **random** = same procedure as in case of Client Hello

• **session ID** = if session ID sent by client is not zero, server will search for previously cached sessions and if a match is found, that session ID will be used; otherwise a new session will be created, i.e., the server will return 0

• **CipherSuite** = single cipher suite selected by server from those proposed by client – if supported, server will agree on client's preferred cipher suite

• **compression method** = compression method selected by server from those proposed by client – if supported, server will agree on client's preferred compression method



Generating Master secret using pre-master secret.
Cipher suite is given as (TLS_RSA_WITH_RC4_128_SHA)

Inside wireshark, we can actually see the list of cipher suites we are sending in the **client-hello** record (strongest and preferred suites are at the top). The one that we will be using can be found in the **server-hello** record. This is also where our client_random and server_random can be found.

Inside the **client-key-exchange** record, we can find our pre-master-secret. Since the ciphersuite we use is TLS_RSA_*, it means that we are using RSA to encrypt the pre-master-secret (no-one but the server at github.com can decrypt this data!).

Decoding TLS_RSA_WITH_RC4_128_SHA

We already know that we are using the cipher suite **TLS_RSA_WITH_RC4_128_SHA.** This means that the pre-master-key is sent over via RSA. But it also means that the actual data encryption is done through **RC4** (which is a commercial version, the "free" version is called **ARCFOUR**, which gives the EXACT same results). The 128 means that we are using a secret key that is 128 bits (16 byte) long, and the SHA part means we are using SHA1 for checking the integrity of our data.

FROM PRE-MASTER SECRET TO MASTER SECRET

First we need to convert our **pre-master secret** to a **master-secret**. If you look closer inside your **SSLKEYLOGFILE**, you will see lines starting with **CLIENT_RANDOM**. These lines contain a client-random value and a master-secret. So it's pretty easy to find the correct master-secret. However, we will do it manually which is possible with the data we've already gathered.

```
// Pre master secret
$pre_master_secret =
hex2bin("03034f855727c944e11c5d74490ce62b550db46a96b32a7be76d68342dcc
7fe9c87026090ebb99245f3ffd13f9ed185a");

// Client and server random values
```

```
$client_random =
hex2bin("52c14c28f2e0af0f3f02228be4b79e0475ba987902f47fce67d2a58a778d
5f6a");
$server_random =
hex2bin("52c14c298bf57e252efc0835a685b875bb9670e7ab2293e03b78381e046d
e736");

// Calculate master secret
$master_secret = prf_tls12($pre_master_secret, 'master secret',
$client_random . $server_random, 48);
```

As we can see, the master secret is created by the pre master secret (in binary form), a literal string **'master secret'**, and the concatenation of the client_random and server_random. The 48 means the master_secret must be 48 bytes long.

The **prf_tls12** function is pretty easy too:

```
function prf_tls12($secret, $label, $seed, $size = 48) {
  return p_hash("sha256", $secret, $label . $seed, $size);
}
```

It just uses the p_hash function, where it will be using sha256 to create our key. No matter which cipher we use for our communication, the generation of the master-secret in TLS1.2 is always using sha256.

Next up, the p_hash function:

```
function p_hash($algo, $secret, $seed, $size) {
    $output = "";
    $a = $seed;

    while (strlen($output) < $size) {
        $a = hash_hmac($algo, $a, $secret, true);
        $output .= hash_hmac($algo, $a . $seed, $secret, true);
```

```
    }

    return substr($output, 0, $size);
}
```

Ok, this looks a bit complex, but basically what it does is generate some output based on the previous output. What it does is explained in http://tools.ietf.org/html/rfc5246#section-5, the actual RFC for TLS 1.2. But it boils down to this:

```
P_SHA256(secret, seed) = HMAC_SHA256(secret, A(1) + seed) +
                         HMAC_SHA256(secret, A(2) + seed) +
                         HMAC_SHA256(secret, A(3) + seed) + ...

   A() is defined as:
      A(0) = seed
      A(i) = HMAC_SHA256(secret, A(i-1))
```

Depending on how much data we need, we have to run this multiple times. Since sha256 returns a hash of 32 bytes (256 bits), and we need 48 bytes for our master-secret, it means that we must run this p_hash function twice so we have 64 bytes. Then we will return 48 bytes and just ignore the last 16 bytes. Every time we add more data, we use the output of the previous round as the input of this round.

After this, we have our actual master-secret. You can verify if this is correct, by checking against the master-secret you can find in the **SSLKEYLOGFILE.**

----------THE END----------