Putting It All Together

Securing a Communication Channel

The definition of a secure channel is one that establishes, between two programs, a data channel that has confidentiality, integrity and authenticity against a computationally-bounded network attacker.

This will involve the use of all concepts learnt so far. Let us see how we can secure a communication channel between Alice and Bob.com.

Step 1: Unilateral Authenticated Key Exchange

Alice and Bob.com will carry out unilateral authenticated key exchange using Bob's private and public key.

After authentication, both Bob and Alice know two randomly selected session keys (k, t), where k is the secret key of a symmetric-key encryption e.g. AES, and t is the secret key of a MAC.

Step 2: Authenticated Encryption

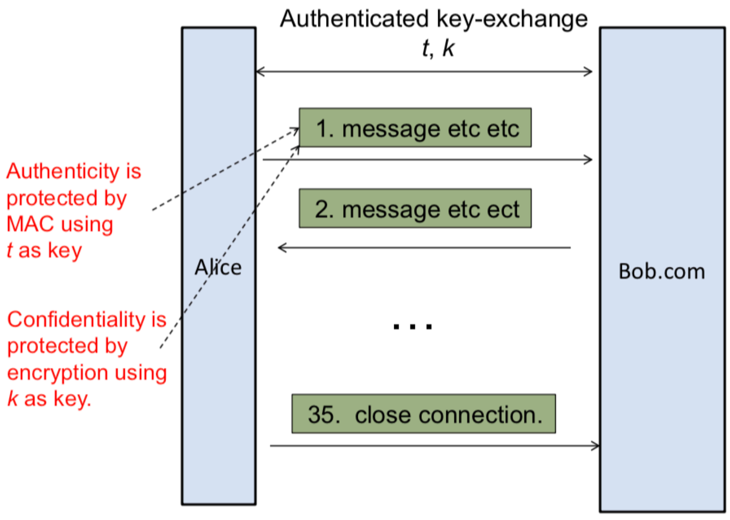
Subsequent communication between Alice and Bob.com will be protected by k, t and a sequence number i.

Suppose m_1, m_2, m_3, … are the sequence of messages exchanged, the actual data to be sent for m_i will be:

$$E_k (i \mathbin{||} m_i) \mathbin{||} MAC (E_k (i \mathbin{||} m_i))$$

where i is the sequence number and || refers to concatenation.

The above is known as "encrypt-then-MAC", while there are other variants of authenticated encryption such as "MAC-then-encrypt" and "MAC-and-encrypt".



Authenticated key-exchange
t, k
1. message etc etc
2. message etc ect
...
35. close connection.
Alice
Bob.com
Authenticity is protected by MAC using *t* as key
Confidentiality is protected by encryption using *k* as key.

There is a need for the sequence number as it helps both sides to detect duplicates, dropped records and out-of-order records.

Use of PKI

For the above, PKI is often employed to distribute the public key. The authenticated key exchange is thus likely to involve certificates. After all, Alice needs to verify that the entity she is communicating with is indeed Bob.com.

HTTPS

HTTPS (Hypertext Transfer Protocol Secure) is widely used to secure Web traffic. It is built on top of SSL (Secure Sockets Layer) / TLS (Transfer Layer Security), i.e. HTTPS = HTTP + SSL. Hence HTTPS is also called HTTP over SSL or HTTP over TLS.

SSL / TLS

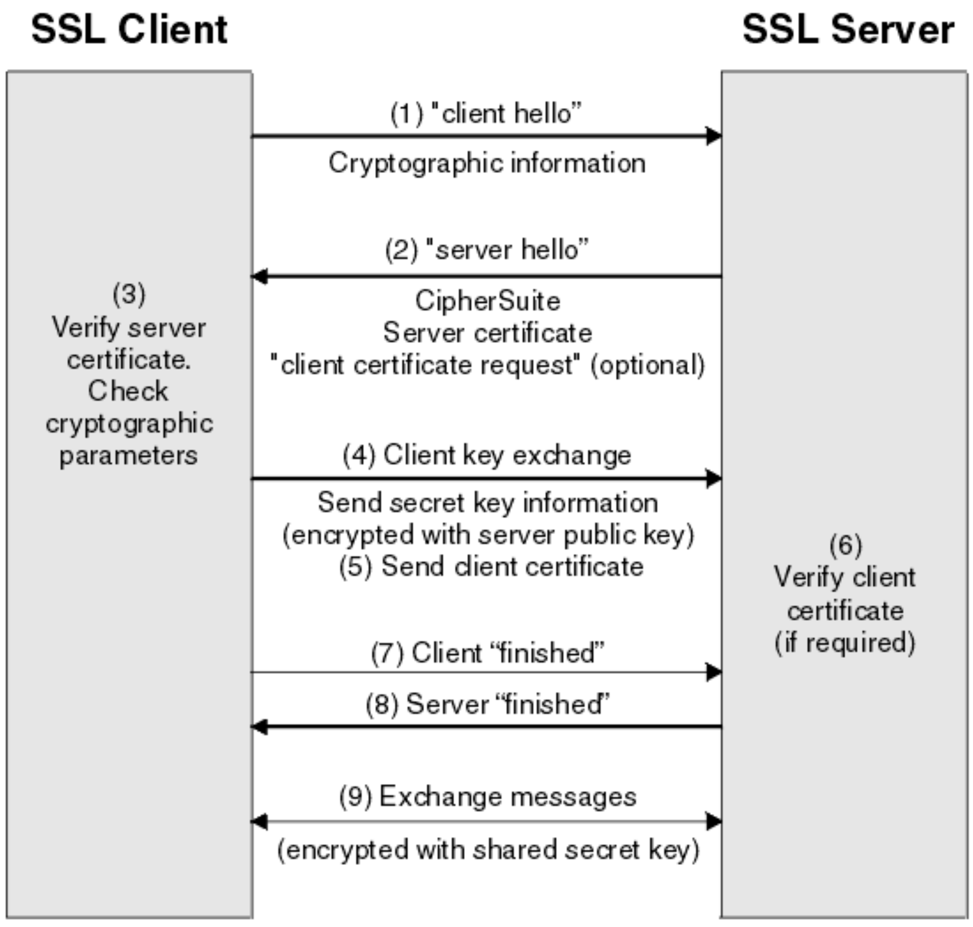
Transport Layer Security (TLS) is a protocol to secure communication using cryptographic means. SSL is the predecessor of TLS: Netscape SSL 2.0. They adopt a similar framework to secure a communication channel, and works in a similar manner to what was described above.

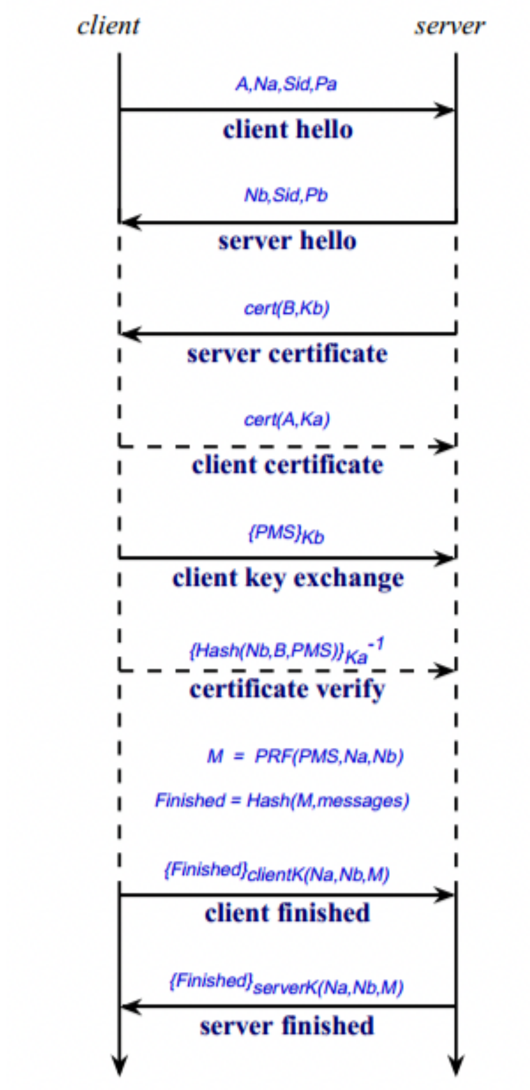
HTTPS

Overview of how HTTPS works:

- Ciphers Negotiation
- Authenticated Key Exchange (AKE)
    - Exchange of session key, which also authenticates the identities of parties involved
- Symmetric-key based secure communication
- Re-negotiation (if necessary)

TLS Handshake (Ciphers Negotiation & Authenticated Key Exchange)


SSL Client
SSL Server
(1) "client hello"
Cryptographic information
(2) "server hello"
CipherSuite
Server certificate
"client certificate request" (optional)
(3) Verify server certificate. Check cryptographic parameters
(4) Client key exchange
Send secret key information (encrypted with server public key)
(5) Send client certificate
(6) Verify client certificate (if required)
(7) Client "finished"
(8) Server "finished"
(9) Exchange messages
(encrypted with shared secret key)



To summarise, these are the steps

1. User asks for certificate — handshake
2. Server / Site sends the certificate — handshake
3. User verifies certificate and gets the public key Ke
4. User will generate a session key pair (k, t) - k is for encryption, t is for authentication
5. User will encrypt his session key pair using public key : E(Ke, (k, t)) and send to Server

This session key pair will expire after a while, and renegotiation is needed. What happens is that steps 1-3 will be skipped, since we already have the certificate and the public key. We just need to redo steps 4 and 5.

Protocol
In computer networking, a protocol is a set of rules for exchanging information between multiple entities. A protocol is often described as steps of actions to be carried out by the entities, and the data to be transmitted.

<u>TLS Renegotiation Attack</u>
We must first understand the TLS renegotiation feature. This feature allows a client and server who already have a TLS connection to negotiate new parameters, generate new keys, and more. This renegotiation is carried out **in the existing TLS connection**.

This renegotiation feature thus creates a problem – the encrypted connection before renegotiation and that after renegotiation can actually be controlled by different parties! To make things worse, web servers will combine the data they receive prior to renegotiation (which is coming from an attacker) with the data they receive after renegotiation (which is coming from a victim).

```
Client                        Attacker                        Server
------                        -------                         ------
                                  <----------- Handshake ---------->
                                  <======= Initial Traffic ========>
<--------------------------  Handshake ============================>
<======================== Client Traffic ==========================>
```

How does this work?

1. To mount this attack, the attacker first connects to the TLS server. A first handshake is done, and session keys are established. He can communicate with the server as much as he wants.
2. When he's ready, he hijacks the client's connection with the server, and when the client tries to establish their connection with the server, the attacker will intercept and proxy the client's traffic over his encrypted channel.
    a. This client hello is in clear, since no public key has been obtained yet.
    b. The attacker then encrypts this client hello using his session keys.
3. To the server, this seems like a renegotiation request, and the session keys generated by the client but redirected to the server by the attacker would seem like they were from the attacker. The server has no way to determine whose keys these are.
4. The attacker gets the acknowledgement and replays it to the client. The client will thus think that everything is okay.
5. The result is that the attacker's initial traffic and the client's subsequent traffic would be combined, with the web server thinking that the initial traffic was also from the client, or that the subsequent traffic is also from the attacker. Usually it is the former that would be more beneficial.

Possible Applications
This can be exploited in a way that is somewhat similar to an injection attack. The attacker can send a partial HTTP request that requests for some resource, but requires certain cookies, which are secret tokens that are sent with any request and are usually used as proof of identity.

Afterwards, the "renegotiation" is handled by the client, and the client's real request is concatenated to the end of the attacker's request, and the client's cookies are used instead.

For example, the attacker may send:
```
GET /pizza?toppings=pepperoni;address=attackersaddress HTTP/1.1
X-Ignore-This:
```

And leave the last line empty without a carriage return line feed. When the client makes his own request:
```
GET /pizza?toppings=sausage;address=victimssaddress HTTP/1.1
Cookie: victimscookie
```

The two requests are merged into:
```
GET /pizza?toppings=pepperoni;address=attackersaddress HTTP/1.1
X-Ignore-This: GET /pizza?toppings=sausage;address=victimssaddress HTTP/1.1
Cookie: victimscookie
```

The server then uses the victim's account to send a pizza to the attacker.

This is another example:
```
GET /path/to/resource.jsp HTTP/1.0
Dummy-Header: GET /index.jsp HTTP/1.0
Cookie: sessionCookie=Token
```

where the red highlighted part is by the attacker, and the green is by the victim.

Notes on the attack
The attacker does not know the session keys that the client has established with the server, nor does he get to see any sensitive information that the client sends directly. It's all encrypted. However, he can exploit side effects of the exchange, such as getting a pizza out of it.

In other words, the renegotiation attack on TLS does not compromise confidentiality. However, it may breach the integrity of the client-server communication.

The second handshake has its key partially encrypted. It contains the session keys encrypted by the client using the server's public key, which is further encrypted by the attacker's current session key pair.

Some applications actually prompt for renegotiation when certain orders are being performed. Those are also possible for attackers to exploit.

Naïve Solutions that Do Not Work
1. Change the handshake message **from client to server**:
    a. Original: "Hello, I want to connect"
    b. New: "Hello, and I want to connect and this is the x handshake"
        i. Where x can be first, second, etc.
    c. Does not work as the attacker can simply change the first to second. This is because the original client hello is in clear to the attacker. It is unsure if the attacker would forward this as well, since technically this step can be skipped for the attacker.
2. Change the handshake message **from server to client**:
    a. Original: "Ok, I am happy to connect. Here is my certificate and other information."
    b. New: "Ok, I am happy to connect. Here is my certificate and other information. This is our x handshake."
        i. Where x can be first, second etc.
    c. Does not work as the attacker can change second to first easily. Unsure if the attacker would forward the initial client hello:
        i. If he forwards it, the reply would be encrypted, and the attacker can easily decrypt it and change x to the correct value before sending it back in clear to the client.
        ii. If he does not forward it, he can just reply the client with a standard response with x being first.

Solutions from Transport Layer that May Work

1. Server can sign the message “Ok, I am happy to connect… This is our x handshake.” using its private key, thus preventing the attacker from modifying it.
2. It can apply MAC using a secret key previously established with the client.

Solutions from Application Layer that May Work

1. Send two GET commands instead of one, where the first GET command is just a dummy operation.
2. The GET command includes (part of) the cookie for the server’s verification.
3. The GET command includes a value that is derived from the cookie, e.g. MAC of the GET command and the secret cookie, for the server’s verification.

However, any remedial solution from the web developer would be messy, as even it if works, the application itself now has to take care of communication security. This is not desirable since it is against the modularity principle of a good system design. That is, communication security is supposed to be handled by the layer below the application layer.

Can’t we just disable renegotiation?
It will affect availability since some applications may need the renegotiation feature of the TLS protocol.

Authentication in Applications
Most web applications perform an initial client authentication via a client’s username / password pair, and then subsequently persist that authentication state with HTTP / eb cookies (i.e. secret server-generated tokens that are automatically sent with subsequent client’s requests to the server).

More will be covered in subsequent topics.