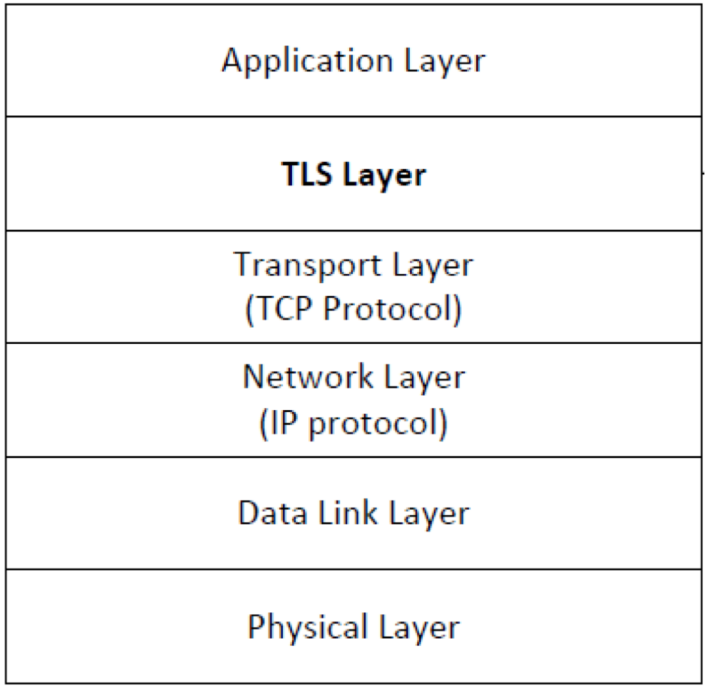
Transport Layer Security

Overview of TLS

- Transport Layer Security (TLS) is a protocol that provides a secure channel between two communicating applications. The secure channel has 3 properties:
  - Confidentiality: Nobody other than the two ends of the channel can see the actual content of the data transmitted.
  - Integrity: Channel can detect any changes made to the data during transmission
  - Authentication: At least one end of the channel needs to be authenticated, so the other end knows who it is talking to.

TLS Layer

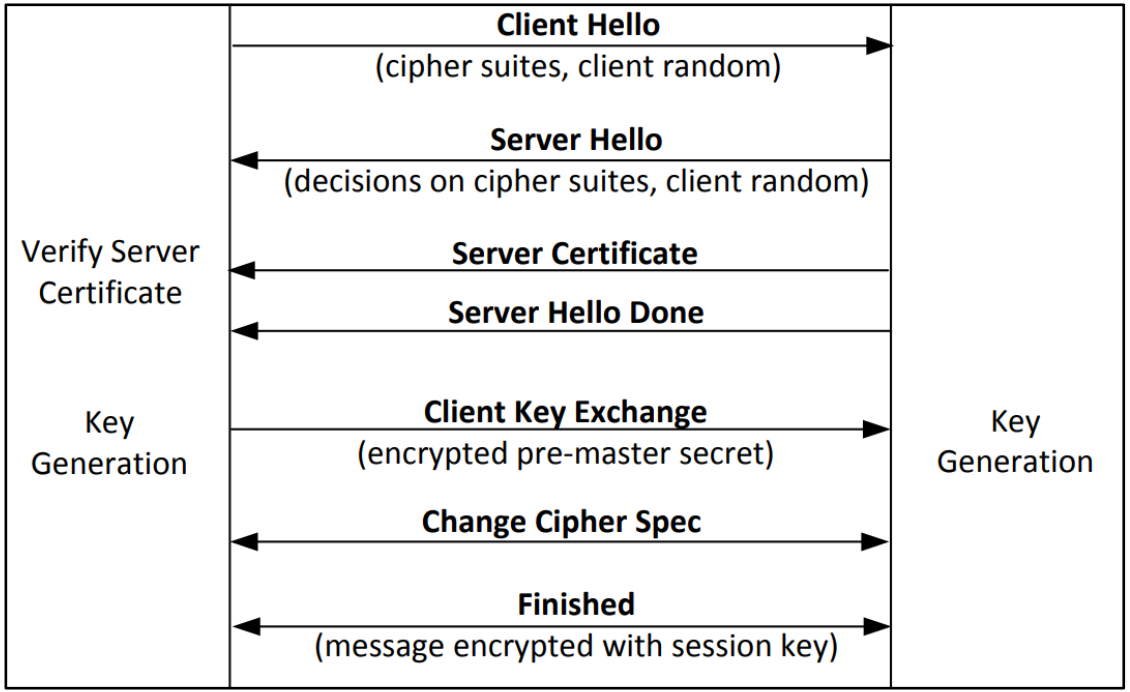
- TLS sits between the Transport and Application layer
  - Unprotected data is given to TLS by Application layer
  - TLS handles encryption, decryption and integrity checks
  - TLS gives protected data to Transport layer



TLS Handshake

- Before a client and server can communicate securely, several things need to be set up first:
  - Encryption algorithm and key
  - MAC algorithm
  - Algorithm for key exchange
- These cryptographic parameters need to be agreed upon by the client and server
- This is the primary purpose of the handshake protocol

TLS Handshake Protocol



Network Traffics During TLS Handshake

Since TLS runs top of TCP, a TCP connection needs to be established before the handshake protocol. This is how the packet exchange looks between a client and server during a TLS handshake protocol captured using Wireshark:

No.	Source	Destination	Protocol	Info
1	10.0.2.45	10.0.2.35	TCP	59930 –> 11110 [SYN] Seq=0 Win=14600 Len=0 MSS=1460...
2	10.0.2.35	10.0.2.45	TCP	11110 –> 59930 [SYN, ACK] Seq=0 Ack=1 Win=14480...
3	10.0.2.45	10.0.2.35	TCP	59930 –> 11110 [ACK] Seq=1 Ack=1 Win=14720 Len=0...
4	10.0.2.45	10.0.2.35	TLSv1.2	Client Hello
6	10.0.2.35	10.0.2.45	TLSv1.2	Server Hello, Certificate, Server Hello Done
8	10.0.2.45	10.0.2.35	TLSv1.2	Client Key Exchange, Change Cipher Spec, Finished
9	10.0.2.35	10.0.2.45	TLSv1.2	New Session Ticket, Change Cipher Spec, Finished

Certificate Verification

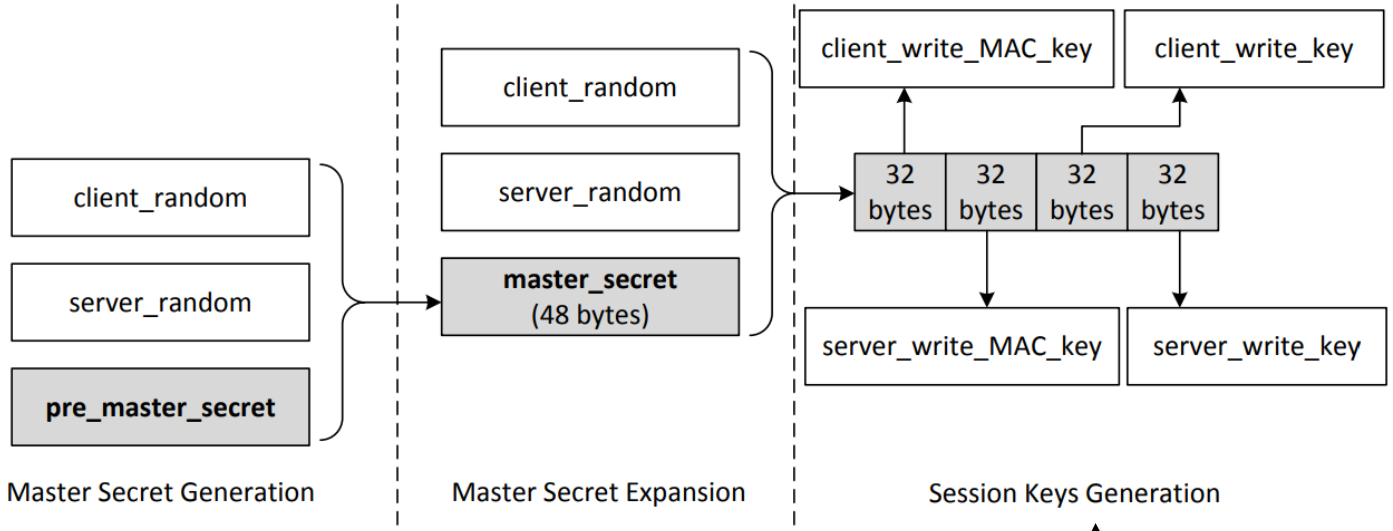
- The client first does a validation check of the certificate
  - Check expiration date, signature validity, etc.
  - Hostname and certificate's common name match
- The client needs to have the singing CA's public-key certificate.

Key Generation and Exchange

- Although public-key algorithms can be used to encrypt data, it is much more expensive than secret-key algorithms.
  - TLS uses PKI for key exchange.
  - After that, server and client switch to secret-key encryption algorithm

- The entire key generation consists of three steps:
  - Step 1: Generating pre-master secret
  - Step 2: Generating master secret
  - Step 3: Generating session keys

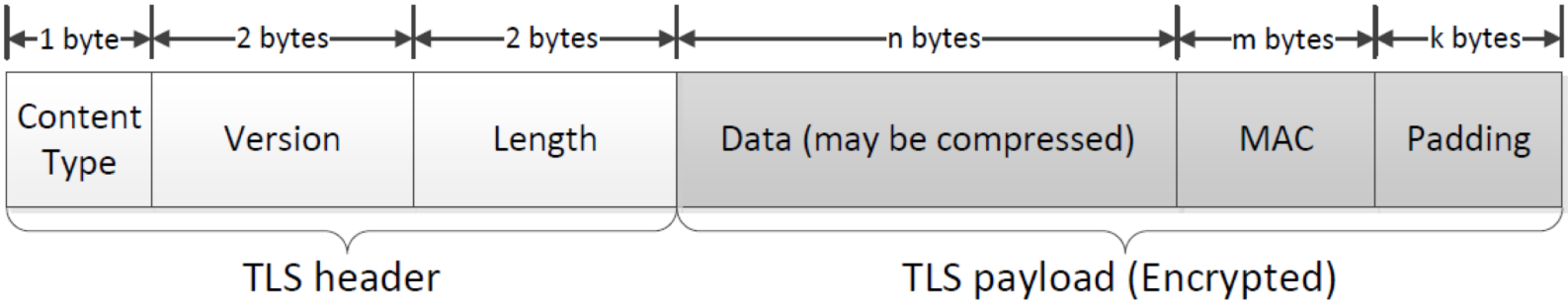
Key Generation and Exchange



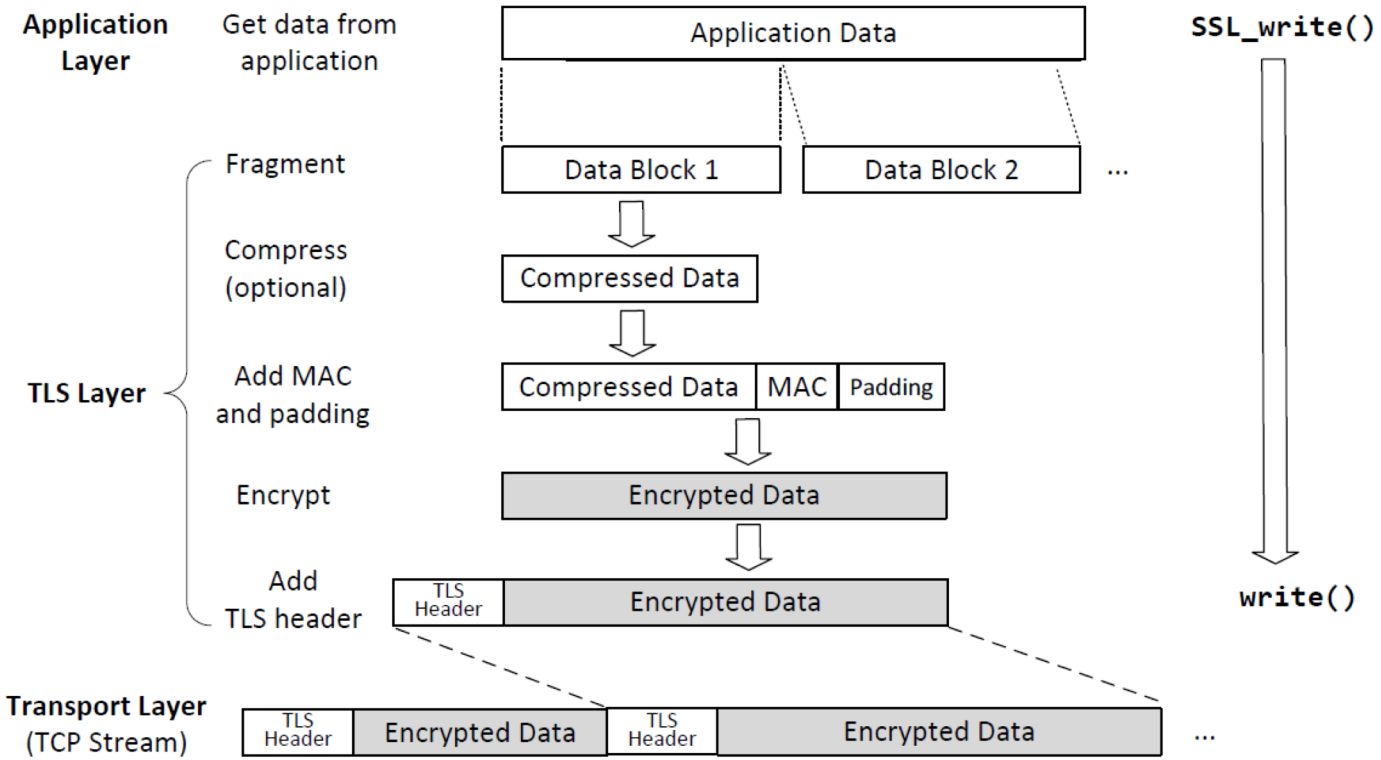
These keys are used to protect an SSL session

TLS Data Transmission

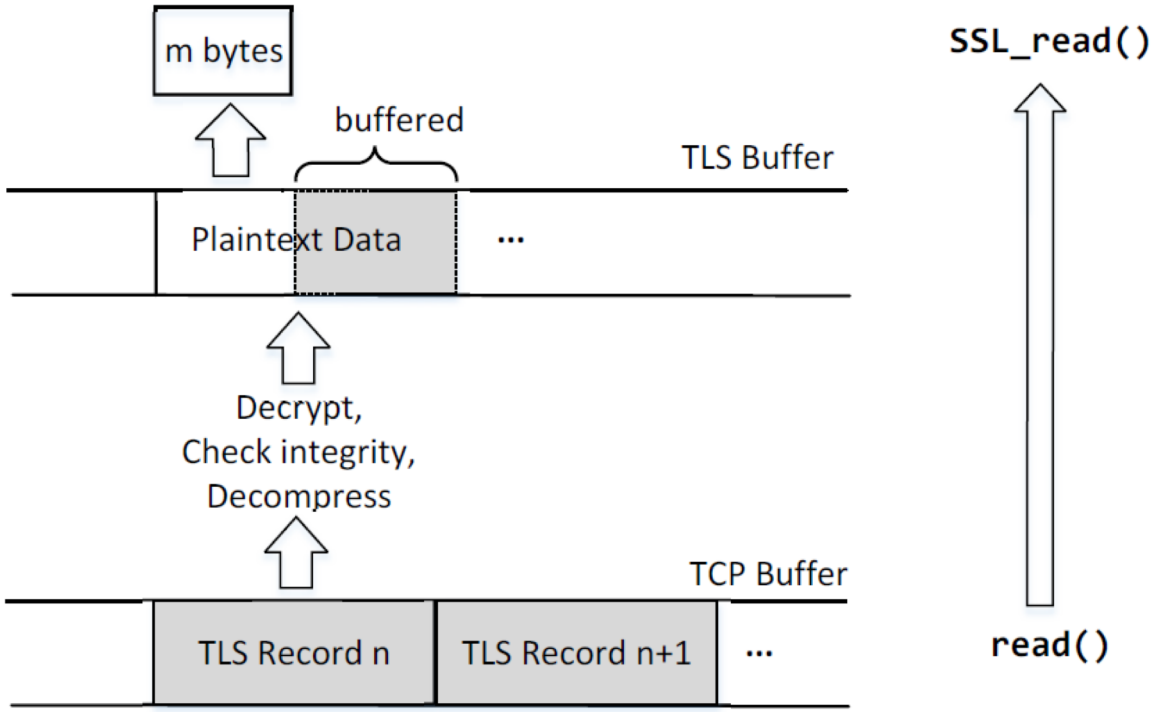
- Once the handshake protocol is finished, client and server can start exchanging data.
- Data is transferred using records.
- Each record contains a header and a payload



Sending Data with the TLS Record Protocol

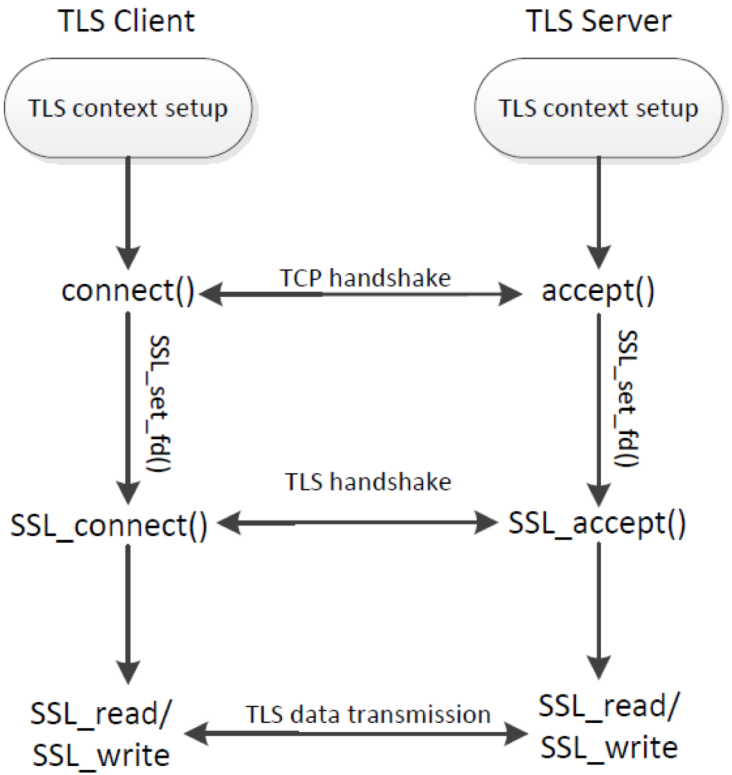


Receiving Data with the TLS Record Protocol



TLS Client Program

TLS Programming : Overall Picture



TLS Client Program: TLS Initialization

- TLS protocol is a stateful protocol
- Create a context data structure
- Create a SSL structure to hold state information

SSL Context: holding SSL configuration →

Holding SSL states →

```
// Step 1: SSL context initialization
SSL_METHOD *meth = (SSL_METHOD *)TLSv1_2_method();
SSL_CTX* ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_load_verify_locations(ctx, NULL, "./cert");

// Step 2: Create a new SSL structure for a connection
SSL* ssl = SSL_new (ctx);
```

TLS Client Program: TLS Initialization (cont'd)

```
// Step 1: SSL context initialization
SSL_METHOD *meth = (SSL_METHOD *)TLSv1_2_method();
SSL_CTX* ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_load_verify_locations(ctx, NULL, "./cert");

// Step 2: Create a new SSL structure for a connection
SSL* ssl = SSL_new (ctx);
```

Should verify server's certificate

Folder containing trusted CA' certificates, such as root CA's certificates.

```
// Step 3: Enable the hostname check
X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
```

Check whether the certificate's subject field matches with hostname.

TLS Client Program: Set Up a TCP Connection

- TLS is primarily built on top of TCP.
- This part is standard.

```
int setupTCPClient(const char* hostname, int port)
{
   struct sockaddr_in server_addr;

   // Get the IP address from hostname
   struct hostent* hp = gethostbyname(hostname);

   // Create a TCP socket
   int sockfd= socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   // Fill in the destination information (IP, port #, and family)
   memset (&server_addr, '\0', sizeof(server_addr));
   memcpy(&(server_addr.sin_addr.s_addr), hp->h_addr, hp->h_length);
   server_addr.sin_port   = htons (port);
   server_addr.sin_family = AF_INET;

   // Connect to the destination
   connect(sockfd, (struct sockaddr*) &server_addr,
           sizeof(server_addr));

   return sockfd;
}
```

TLS Client Program: Initiate TLS Handshake

Establish the SSL session on top of an established TCP connection

```
SSL* ssl = setupTLSClient(hostname);
int  sockfd = setupTCPClient(hostname, port);

SSL_set_fd(ssl, sockfd);
int err = SSL_connect(ssl);
```

Initiate the TLS Handshake protocol

TLS Client Program: Send/Receive Data

- We construct a simple HTTP GET request, and print out the reply from the web server.

Send data

Send data

```
char buf[9000];
char sendBuf[200];

sprintf(sendBuf, "GET / HTTP/1.1\nHost: %s\n\n", hostname);
SSL_write(ssl, sendBuf, strlen(sendBuf));

int len;
do {
     len = SSL_read (ssl, buf, sizeof(buf) - 1);
     buf[len] = '\0';
     printf("%s\n",buf);
} while (len > 0);
```

TLS Client Program: Set Up Certificate Folder

- We need to gather some trusted CA certificates and store them in the "./cert" folder:
- Let's see what certificates are need for verifying google.com's certificate:

```
$ openssl s_client -connect www.google.com:443
...
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google
    Inc/CN=www.google.com
   i:/C=US/O=Google Inc/CN=Google Internet Authority G2
 1 s:/C=US/O=Google Inc/CN=Google Internet Authority G2
   i:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
 2 s:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
   i:/C=US/O=Equifax/OU=Equifax Secure Certificate Authority
```

We need to have this certificate, or we will not be able to verify Google's certificate

We can export Equifax's certificate from a browser, and save it in ./cert.

TLS Client Program: Set Up Certificate Folder

- When TLS tries to verify a certificate, it generates a hash from the issuer's identity information.
- The hash value is used as part of the filename to find the issuer's certificate.

Generate the hash using the subject field of the certificate

```
$ openssl x509 -in DSTRootCAX3.pem -noout -subject_hash
2e5ac55d

$ ln -s DSTRootCAX3.pem 2e5ac55d.0
$ ls -l

lrwxrwxrwx 1 ... 2e5ac55d.0 -> DSTRootCAX3.pem
lrwxrwxrwx 1 ... 578d5c04.0 -> EquifaxSecureCA.pem
lrwxrwxrwx 1 ... 9f8b5587.0 -> modelCA_cert.pem
lrwxrwxrwx 1 ... b204d74a.0 -> VeriSignClass3CA-G5
-rw-r--r-- 1 ... DSTRootCAX3.pem
-rw-r--r-- 1 ... EquifaxSecureCA.pem
-rw-r--r-- 1 ... modelCA_cert.pem
-rw-r--r-- 1 ... VeriSignClass3CA-G5
```

TLS Client Program: Testing

- If everything is set up correctly, we should be able to see an HTML page from the web server.
- However, if we did not setup the certificates properly, we are likely to see this error:

```
3085014664:error:14090086:SSL routines:ssl3_get_server_certificate:
certificate verify failed:s3_clnt.c:1258:
```

- Many reason can trigger this error such as an expired certificate, corrupted certificate etc.

Use Our Client Program to Conduct an MITM Experiment

Experiment: Verifying Server's Hostname

We design an experiment to show how important it is to verify server's hostname. We slightly modify our client program, so we can print out more information during runtime.

```
meth = (SSL_METHOD *)TLSv1_2_method();
ctx = SSL_CTX_new(meth);

SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback); ❶
SSL_CTX_load_verify_locations(ctx, NULL, "./cert");
ssl = SSL_new (ctx);

// Enable the hostname check
X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);  ❷
X509_VERIFY_PARAM_set1_host(vpm, hostname, 0); ❸
```

We add a callback function here. It will be triggered every time a certificate is verified.

Experiment Callback Function

```
int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx)
{
    char  buf[300];

    X509* cert = X509_STORE_CTX_get_current_cert(x509_ctx); ❹
    X509_NAME_oneline(X509_get_subject_name(cert), buf, 300);
    printf("subject= %s\n", buf);      ❺

    if (preverify_ok == 1) {
       printf("Verification passed.\n");
    } else {
       int err = X509_STORE_CTX_get_error(x509_ctx);
       printf("Verification failed: %s.\n",
          X509_verify_cert_error_string(err)); ❻
    }
}
```

Get and print out the certificate's subject information

Print out the verification result

Experiment: Man-In-The-Middle Attack

- We simulate a DNS Cache poisoning attack. So, every time users want to visit www.facebook.com, they will go to www.example.org
- Instead of launching a real DNS attack, we manually add an entry to the /etc/hosts file:

www.example.org's IP address

```
93.184.216.34    www.facebook.com
```

- First we try to visit Facebook using our modified client program. But, we comment out the lines that conduct hostname check.

```
$ client www.facebook.com 443
```

Due to the "attack", we will actually visit www.example.org.

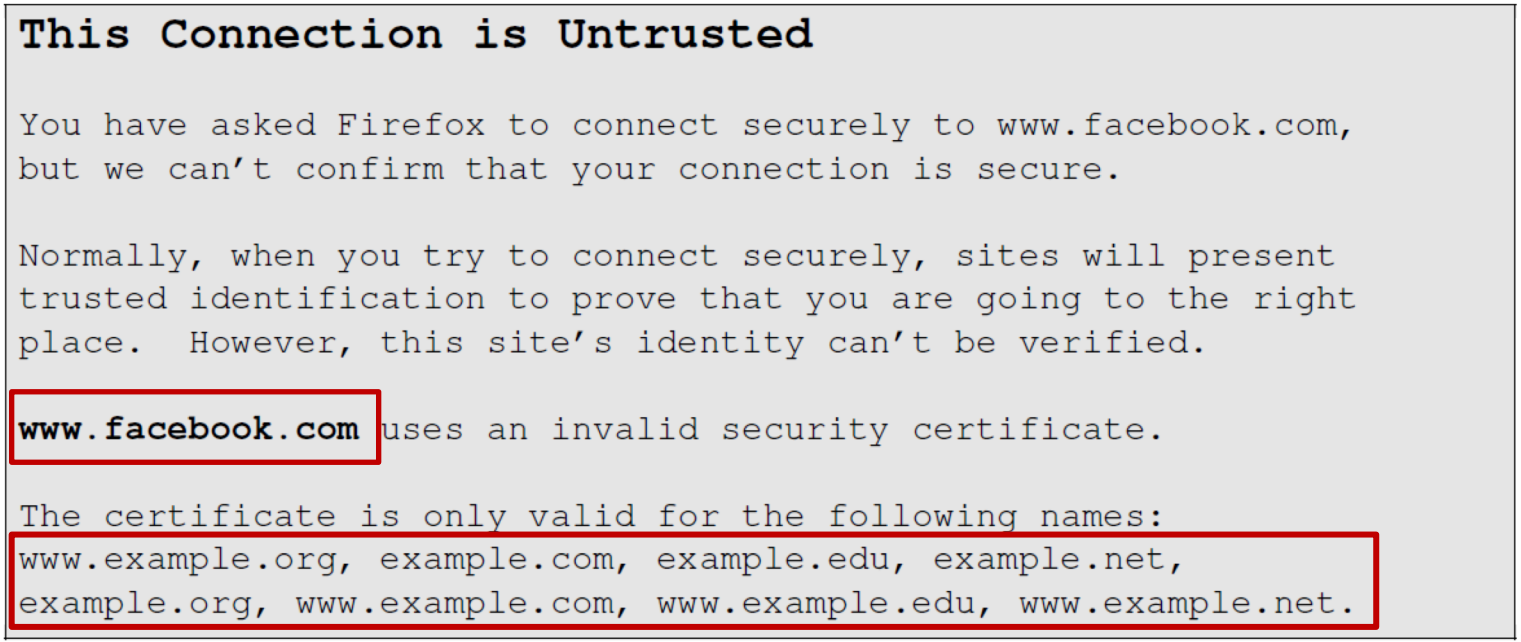
Experiment: Man-In-The-Middle Attack

- All certificate verifications are successful.
- MITM attack is successful.

Running result

```
$ client www.facebook.com 443
subject= ... /CN=DigiCert High Assurance EV Root CA
Verification passed.
subject= ... /CN=DigiCert SHA2 High Assurance Server CA
Verification passed.
subject= ... /CN=www.example.org
Verification passed.
SSL connection is successful
SSL connection using ECDHE-RSA-AES128-GCM-SHA256
```

Running a Real Client (Browser)



Hostname match failed

Verifying Server's Hostname

- Let us add the hostname check back to our own code

```
$ client www.facebook.com 443
subject= /C=US/ST=California/L=Los Angeles/O=Internet Corporation for
Assigned Names and
Numbers/OU=Technology/CN=www.example.org
Verification failed:  Hostname mismatch.  ❶
subject= ... /CN=DigiCert High Assurance EV Root CA
Verification passed.
subject= ... /CN=DigiCert SHA2 High Assurance Server CA
Verification passed.
subject= ... /CN=www.example.org
Verification passed.                                    ❷
SSL connection is successful                            ❸
SSL connection using ECDHE-RSA-AES128-GCM-SHA256
```

Now we can detect the mismatch

We should abort the program, instead of continuing with the SSL connection.

TLS Server Program

Create a simple HTTPS server

TLS Server Program: Setup

```
// Step 1: SSL context initialization
meth = (SSL_METHOD *)TLSv1_2_method();
ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);

// Step 2: Set up the server certificate and private key
SSL_CTX_use_certificate_file(ctx, "./bank_cert.pem",
                             SSL_FILETYPE_PEM);
/* SSL_CTX_use_certificate_chain_file(ctx,
                             "./bank_chain_cert.pem"); */
SSL_CTX_use_PrivateKey_file(ctx, "./bank_key.pem",
                             SSL_FILETYPE_PEM);

// Step 3: Create a new SSL structure for a connection
ssl = SSL_new (ctx);
```

Will not verify the client's certificate

Server's certificate

Server's private key

TLS Server Program: TCP Setup

This program creates a TCP socket, binds it to a TCP port (4433) and marks the socket as a passive socket. This is quite standard.

```
int setupTCPServer()
{
    struct sockaddr_in sa_server;
    int listen_sock;

    listen_sock= socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    memset (&sa_server, '\0', sizeof(sa_server));
    sa_server.sin_family      = AF_INET;
    sa_server.sin_addr.s_addr = INADDR_ANY;
    sa_server.sin_port        = htons (4433);
    bind(listen_sock, (struct sockaddr*)&sa_server,
    sizeof(sa_server));
    listen(listen_sock, 5);
    return listen_sock;
}
```

TLS Server: Handshake & Data Communication

Conduct TLS handshake with the client

We can now use this established SSL session to conduct data communication

```
while (1) {
  int sock = accept(listen_sock, (struct sockaddr*)&sa_client,
  &client_len);
  if (fork() == 0) { // The child process
     close (listen_sock);

     SSL_set_fd (ssl, sock);
     int err = SSL_accept (ssl);
     CHK_SSL(err);
     printf ("SSL connection established!\n");

     processRequest(ssl, sock);
     close(socket);
     return 0;
  } else { // The parent process
      close(sock);
  }
}
```

TLS Server Program: Data Transmission

- Logic for sending/receiving data is the same as the client program.
- We simply send an HTTP reply message back to the client.

```
void processRequest(SSL* ssl, int sock)
{
    char buf[1024];
    int len = SSL_read (ssl, buf, sizeof(buf) - 1);
    buf[len] = '\0';
    printf("Received: %s\n",buf);

    // Construct and send the HTML page
    char *html = "... (omitted) ...";
    SSL_write(ssl, html, strlen(html));
    SSL_shutdown(ssl);  SSL_free(ssl);
}
```

Summary

- TLS Protocol
- Write a simple TLS client program
- Use the client program to understand how MITM attacks are defeated
- Write a simple TLS server program