SSL/TLS SECURITY AND TROUBLESHOOTING

Aditya Lad
Associate Principal Engineer
EMC
Aditya.lad2@emc.com

Prasoon Dwivedi
Software Engineer
mitprasoon@gmail.com

Table of Contents

Introduction

We live in a world of digital communication and cryptography has become an essential part of it. The importance of cryptography and encrypted communication was highlighted best in World War 2, when allied cryptographers were able to break the encryption techniques used by the axis powers. The stories are glorious and workings of ENIGMA still fascinate crypto-scientists because it helped changing the course of World War 2. The attempts to break an encrypted communication have existed since the beginning of encryption. The worst nightmare of a user of crypto services is that someone super smart has secretly found a way to read their encrypted messages. The vastness of attacks possible today on a crypto-based eco system makes it tough to understand and evaluate the practical risk involved. It is not a surprise: it is challenging even for security experts to keep up with new forms of crypto attacks, understand their complexity and working, and evaluate the practical risks involved. Although the science of encryption-decryption commonly known as cryptography is very old and detailed, we will cover brief parts of it that are related to SSL.

As far as modern day cryptography is concerned, SSL/TLS (Secure Socket Layer/Transport Layer Security) is a widely used protocol and a preferred way for encrypting network communication between two systems. The SSL/TLS system has existed since the mid-90s and has undergone a number of changes for better security in times where the computing power of systems has risen exponentially. Even though the SSL/TLS eco-system is widely used and has been there for some time, its internal workings can still be called complicated and a beginner always has to struggle his/her way out while solving an SSL-related problem. A very recent example of such a situation was the Heartbleed bug where the entire internet – especially the IT world – (i.e. System admins, developers, testers) seemed to be in a state of chaotic confusion on what is the source and real extent of the problem. This article presents a perspective on SSL security that helps a day to day developer or a tester become familiar with SSL/TLS jargon and to know what, how, and where to look while solving an SSL/TLS related problem. The focus of this article is not to explain SSL/TLS or cryptography in an academic sense, but to present a birds-eye view of the intricacies involved and important points from a practical IT perspective.

History of SSL

The development of SSL began in the early 1990s by Netscape and the first draft was submitted for SSL v2.0 in 1995. SSL v2.0 had major security flaws which led to the making of SSL v3.0. The draft for SSL v3.0 was submitted to the IETF in 1996. In Netscape's words[1], SSL v3.0 is a security protocol that prevents eavesdropping, tampering, or message forgery over the Internet. The IETF published RFC 6101[2] (Request for Comment) as specification for SSL v 3.0. SSL began to be called TLS and the next version of TLS came in 1999 with RFC 2246[3]. In a nutshell, SSL v 3.0 and TLS 1.0 do not have differences that a day to day developer has to be concerned with, but it is better to use TLS 1.0. The next version of TLS which is TLS 1.1 came into existence in 2006 and is defined in RFC 4346[4]. TLS 1.1 has improvements over TLS 1.0. The next version, TLS 1.2, was released in 2008 and is defined through RFC 5246[5]. TLS 1.2 has major changes since TLS 1.1 and it includes support for newer and more secure cryptographic algorithms. TLS 1.3 is still in draft state. RFC 6176[6] has updates for all the SSL/TLS versions and the RFCs 2246 (SSL V 3.0), 4346 (TLS 1.1), 5246[5] (TLS 1.2).

Though we may use SSL or TLS interchangeably in this article, it does not mean we are referring to a specific version but the entire SSL/TLS protocol collectively.

Major versions of SSL/TLS and highlights

Figure 1 displays the timeline of released SSL/TLS versions. SSL/TLS has undergone a lot of changes since its inception and is now being used to secure a number of application layer protocols. For an average user (i.e. IT admin or a software developer), the features and changes can be a little overwhelming. A lot of changes are related to internal workings, better security, and stronger cryptography along with improvements on older designs. For an average software developer who just wants to secure his application layer, the changes in SSL/TLS do not mean a large change in the application behavior.

Figure 1: Timeline of SSL/TLS versions

Until the discovery of the POODLE (Padding Oracle On Downgraded Legacy Encryption) vulnerability, SSL v3 was a fairly popular protocol, but post-POODLE, SSL v3 comes under the insecure category. Both SSL v3 and TLS 1.0 which are not very different from each other are vulnerable to the CBC mode attacks. TLS 1.1 has protections for attacks against the CBC mode and is considered a secure protocol. TLS 1.2 is the best and latest option available.

A report[7] by SSLLabs presented in Black Hat 2010 showed the adoption and statistical usage of SSL protocols over the internet. This is an excellent report for someone who is interested in knowing the state of SSL issues affecting the internet in general. Considering the timeline of TLS version release, the next figure (Figure 2) from the report gives an approximate idea of how well the internet adopts an SSL protocol. Though the data is from 2010, it may indeed come as a surprise that almost 50% of the websites covered in the report were still using SSL version 2.

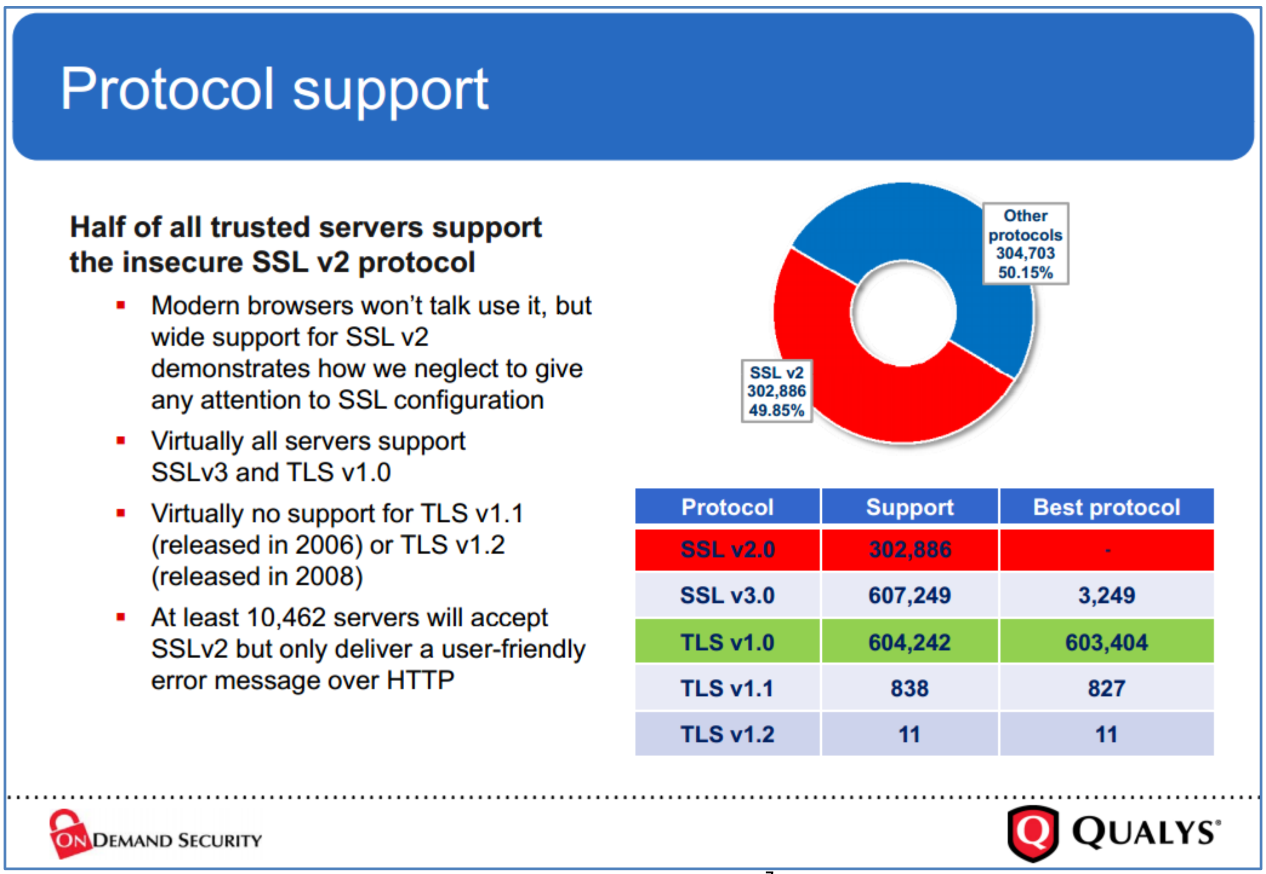


Figure 2: SSLLabs report[7]

Confidentiality, Integrity, and Authentication in TLS

There are three main aspects to SSL/TLS which provide security over the network. There is nothing new about confidentiality, integrity, and authentication and these are the pillars of any secure communication. There are a few things one must keep in mind while working with networks. There is ALWAYS a possibility of someone snooping on your communication and it can be hard to detect it, hence we always need to ensure confidentiality, integrity, and authentication in our communications. Simply put:

Confidentiality – No one except the sender and receiver should be able to decrypt the messages.

Integrity – If someone other than the sender tries to change the content of the message, the receiver should be able to detect it.

Authentication – Sender and receiver (if required) should be able to correctly authenticate each other.

Although we explain cipher-suites in later sections, it is important to understand these three pillars in a practical way by taking the example of an SSL/TLS cipher-suite. A cipher-suite is a collection of different ciphers that are used in an SSL/TLS communication. Let's say we have an existing SSL/TLS communication where the negotiated cipher-suite is **TLS_DHE_RSA_WITH_AES_128_CBC_SHA**.

Confidentiality with Encryption

Encryption algorithm (like **AES_128_CBC** in TLS_DHE_RSA_WITH_**AES_128_CBC**_SHA) of the cipher suite negotiated during SSL handshake is used to encrypt the application data transferred between the server and the client. Using the pre-master secret and random values, a master secret is generated. Using a Pseudo Random Function and the master secret, two keys are generated for server and client respectively, server write key and client write key. The server encrypts the application data using server write key and sends it to the client. This encrypted data can be decrypted only by using the server write key. In the same way client encrypts the application data using client write key and sends it to the client. This encrypted data can be decrypted only by using the client write key.

Integrity using MAC

The MAC algorithm (like **SHA** stands for SHA-1 in TLS_DHE_RSA_WITH_AES_128_CBC_**SHA**) defined in the negotiated cipher suite is used to provide message integrity. For this purpose two MAC keys are also calculated along with the client and server write keys: one for the server, the other for the client. Both server and client are aware of each other's MAC keys. The sender calculates the MAC using its keys and sends it to the receiver along with the application data after encrypting both data and MAC. Upon receiving the package (encrypted data and MAC), the receiver decrypts the data and calculates the MAC using server MAC key. The receiver then validates the integrity of the message by matching the received MAC and calculated MAC.

Authentication with Certificates

Authentication in SSL/TLS (the **RSA** in TLS_DHE_**RSA**_WITH_AES_128_CBC_SHA is responsible for certificate authentication here) is achieved by the use of public key certificates. During SSL handshake the server presents its public key certificate to the client for identity verification. The negotiated cipher suite and the extensions define the exact method with which server authentication is performed by using certificates. Generally, authentication using certificates is performed by validating the digital signatures present in the certificates.

Client authentication by server is optional and happens only if the server requests it. Like server authentication, the client authentication is also dependent on the negotiated cipher suite and extensions during handshake.

Certificates are important components of SSL and caution must be exercised while configuring and installing them. It is recommended to configure a SSL server with multiple type certificates with public key and corresponding private key for interoperability. For SSL server deployments it is highly advisable to use a CA-issued certificate which publishes the revocation information in a Certificate Revocation List (CRL). Self-signed certificates are a strict no-no, especially if the communication is over the Internet.

The client trusts the server on the basis of policies, procedures, and security controls used to issue server public key certificate. The certPolicies extension of X.509 v3 certificate is used to represent these policies, procedures, and security controls. For details, refer RFC5280[8] and RFC6818[9].

Authentication with PSK

In cases where certificates are not used for authentication, a pre-shared key or secret is used for authentication. However, this adds risks in terms of key management and security of the pre-shared key itself. The pre-shared key needs to be shared manually to both client and servers. If you are interested in knowing more about the pre-shared key cipher-suites, RFC5487[10] is a recommended read.

Anatomy of SSL/TLS communication at the Packet level

For understanding and mitigating SSL/TLS related issues, bugs, and vulnerabilities it is important to understand how SSL/TLS works in a practical way. Below is the summary of the working of SSL/TLS protocol based upon RFC 5246 (TLS 1.2). Figure 3 shows the position of SSL/TLS protocol relative to the TCP/IP suite.

SSL/TLS protocol was developed to provide security between sockets at transport layer and the applications accessing these sockets to access the network.

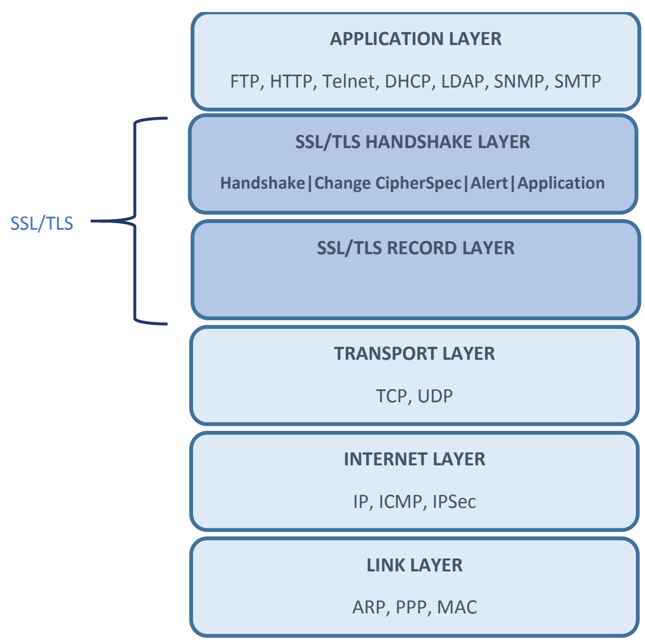


Figure 3: SSL/TLS relative to the TCP/IP layer

Continuing the practical understanding of SSL/TLS, we take the example of a simple HTTPS session captured using the network capture tool, Wireshark. The entire conversation captures 15 packets, which include the initial TCP 3-way handshake, followed by SSL/TLS handshake sequence and encrypted data exchanges. In Figure 4, the topmost row shows the description of the column. The leftmost column is the serial number of the packet, followed by source IP address which is our client browser machine (192.168.32.1), destination IP address which is the SSL server (192.168.32.146), the protocol identified, packet length, and general information on what is inside the packet.

1. **TCP Handshake [Packet #1-3, Figure 4]**
   The first three packets display the standard TCP handshake made by the browser with the SSL web server. This is a standard way that a TCP connection is made between two computers and it is not related to SSL.

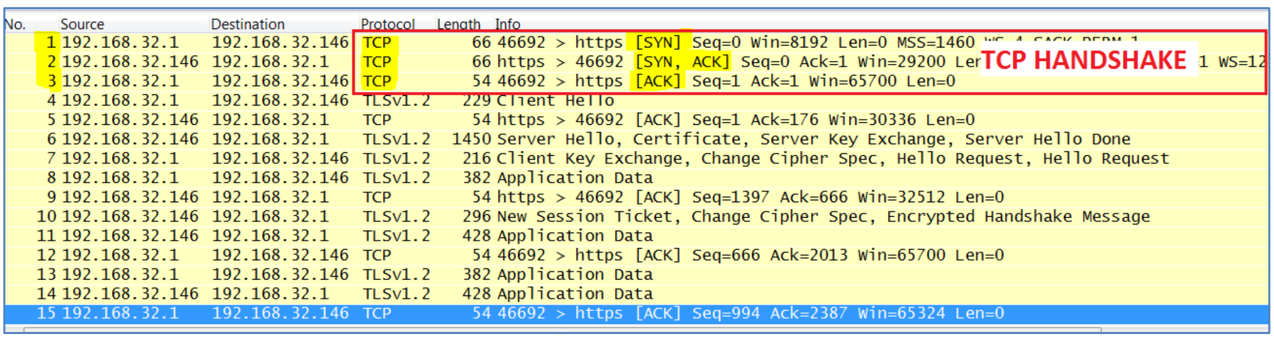


Figure 4: Initial TCP 3-way handshake

2. **Client Greetings to the server [Packet #4-5, Figure 5]**

   The 4th packet starts the SSL protocol with the client (browser) sending a Client Hello message to the SSL server. The ClientHello is a way for the client to greet the server and it contains important details related to the Client's SSL choice like the TLS version it wants to use, random value, session values, supported ciphers, supported compression methods, etc. Figure 5 also highlights some of the attributes sent in the Client Hello message. The 5th packet is an ACK (acknowledgement) packet from the server in response to the ClientHello.

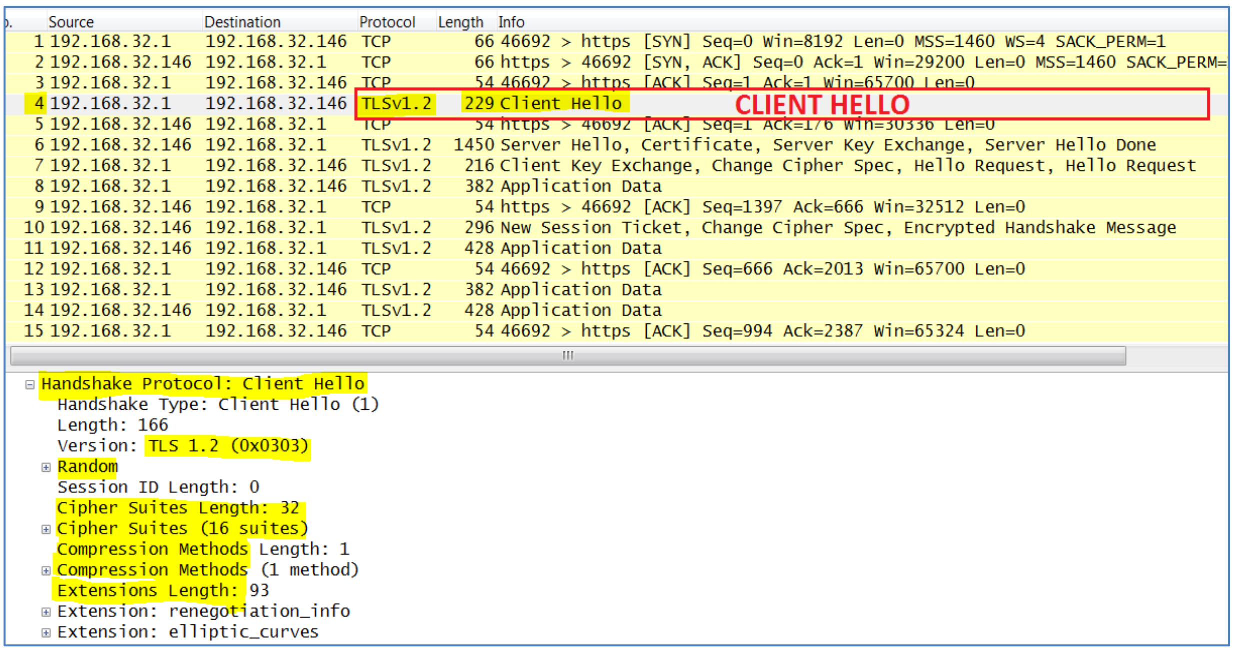


Figure 5: Client Hello message

3. **Server Greetings [Packet #6, Figure 6]**

The 6th packet is the ServerHello message sent by the SSL server and it contains choices, attributes, and certificate sent by the server along with the usage of server key exchange mechanism.

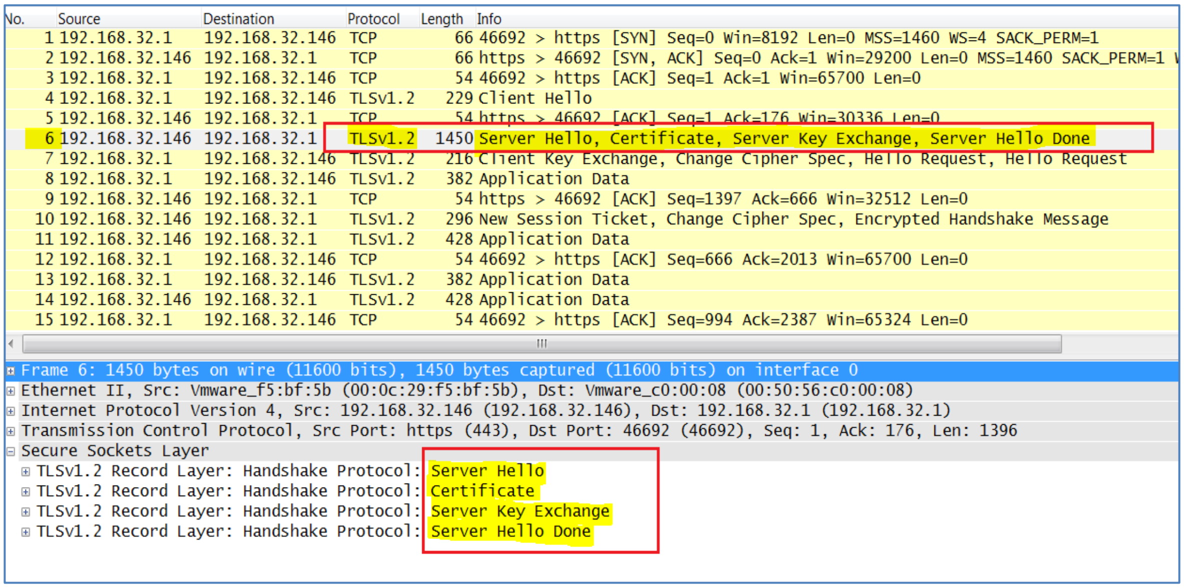


Figure 6: Server Hello message

Figure 7 additionally displays the Server Hello fields in detail and the cipher-suite chosen by the server for the SSL communication. The fields show what version of TLS the server is going to use, whether it supports compression (which is null), and details about the extensions. Note, the choice of CBC mode ciphers along with a vulnerable SSL/TLS version like SSLv3 or TLS 1.0 can make scanners report this connection, vulnerable to attacks prevalent against CBC mode ciphers (i.e. BREACH, POODLE etc.). The compression method value if enabled also makes the server vulnerable to attacks like CRIME and TIME.

```
⊟ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 74
  ⊟ Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 70
      Version: TLS 1.2 (0x0303)
    ⊞ Random
      Session ID Length: 0
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
      Compression Method: null (0)
      Extensions Length: 30
    ⊞ Extension: renegotiation_info
    ⊞ Extension: ec_point_formats
    ⊞ Extension: SessionTicket TLS
    ⊞ Extension: next_protocol_negotiation
⊟ TLSv1.2 Record Layer: Handshake Protocol: Certificate
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 965
  ⊟ Handshake Protocol: Certificate
      Handshake Type: Certificate (11)
      Length: 961
      Certificates Length: 958
    ⊟ Certificates (958 bytes)
        Certificate Length: 955
      ⊞ Certificate (id-at-commonName=nginx-dev,id-at-organizationalUnitName=Nginx dev,id-at-organizationName=Test Nginx,id-at-loc
⊟ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
```

Figure 7: Details of the Server Hello message

4. **Certificate verification, Client Key exchange [Packet #7, Figure 8]**

The 7th packet sent by the client has the Client Key exchange, Change cipher spec protocols.

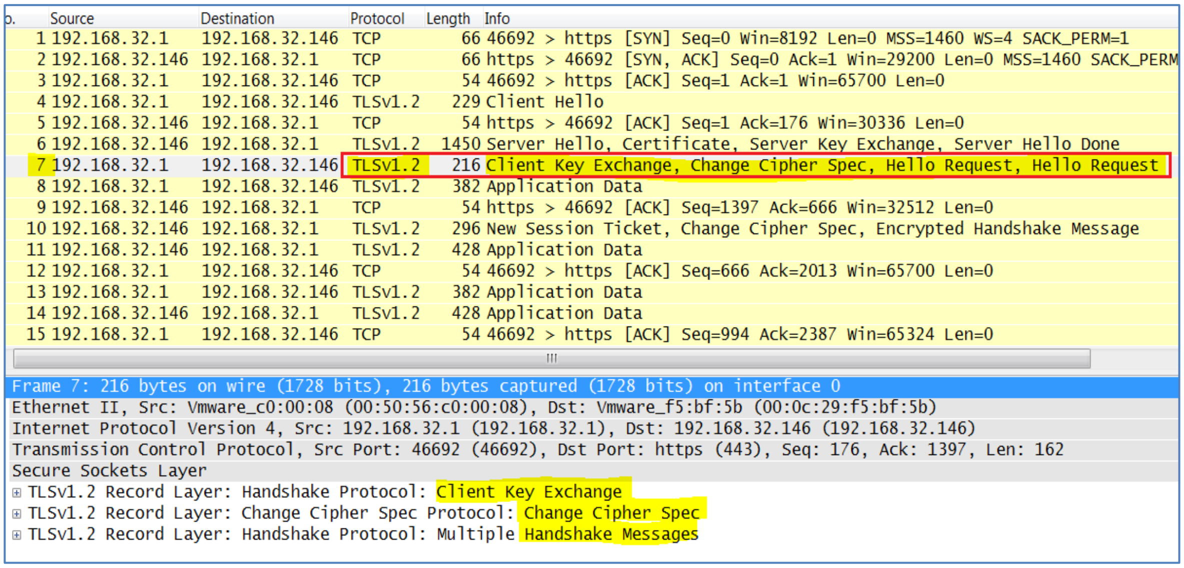


Figure 8: Certificate verification by Client

5. **Client starts encrypting data [Packet #8-9, Figure 9]**

The 8th and 9th packet is where the client starts sending encrypted Application Data to the server.

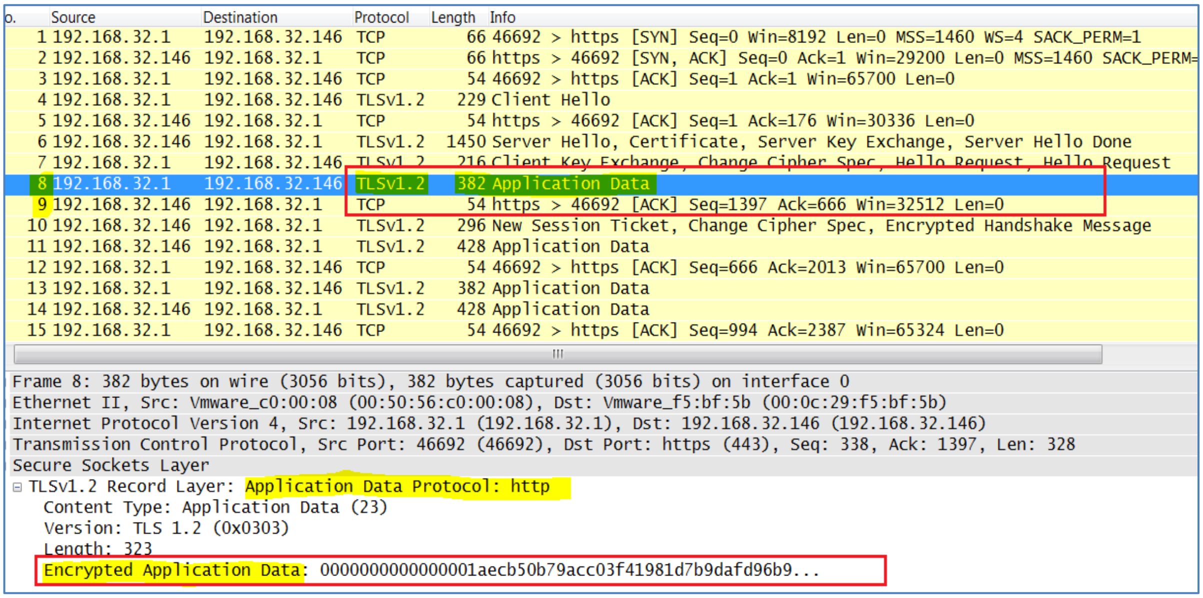


Figure 9: Client encryption begins

6. **Server Changecipherspec and encryption [Packet #10 onwards, Figure 10]**
   The 10th packet is where the server generates a new session ticket, change cipher spec, and starts encrypting data from its side. The Application data protocol comes into the picture now and both the client and server start exchanging encrypted data.

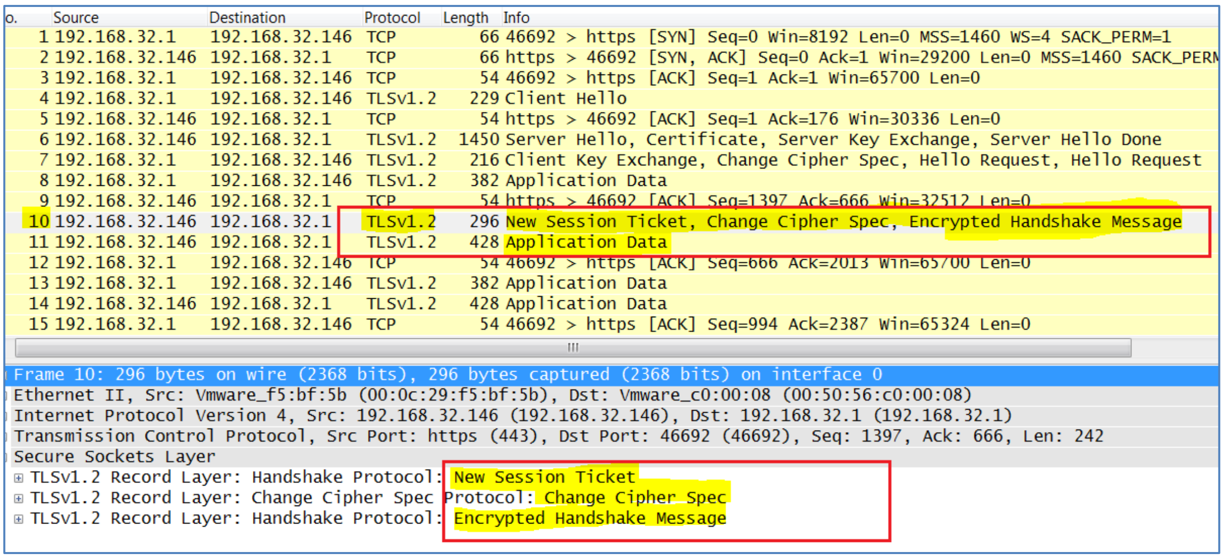


Figure 10: Server change cipher spec

SSL/TLS Protocol structure

Now that we have a basic idea of the anatomy of an SSL/TLS communication, the following defines briefly the important structural and commonly referred parts of the SSL/TLS protocol along with images of packet captures to show how to locate these fields in a network packet.

TLS Record Protocol

The record layer communicates directly with the transport layer. This sub layer of the SSL/TLS protocol is responsible for performing fragmentation of messages into manageable blocks, compression, encryption, and then transmitting the blocks to the lower layer. This layer also receives the data from transport layer; decompresses and decrypts it; rearranges the blocks; and sends them to the higher-level application protocols.

TLS Handshaking Protocols

This is a layered protocol containing four sub protocols. At each layer, there are fields for version, content type, length, and content. The four sub protocols are:

1. Handshake protocol
2. Change Cipher Spec Protocol
3. Application Data Protocol
4. Alert Protocol

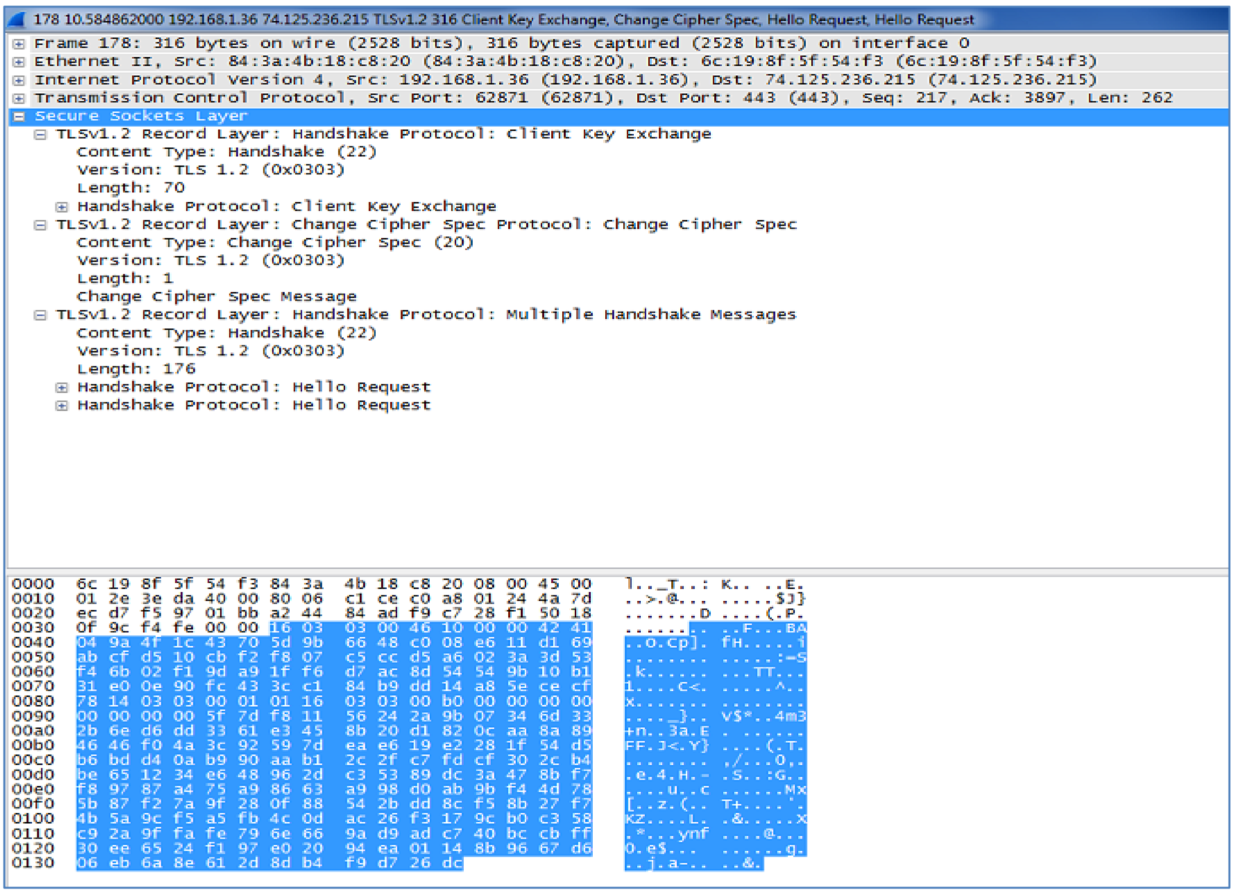


Figure 11: SSL/TLS Sub protocols

1. Handshake Protocol (ClientHello, ServerHello, Certificate, ServerKeyExchange, CertificateRequest, ServerHelloDone)

The Handshake protocol is used to negotiate attributes for a secured session between a client and the server. The client's way of greeting the server is through the ClientHello message while the server's way of replying to the client greeting is the ServerHello message.

- **ClientHello** - This message is sent from the client to the server whenever it tries to connect to the server or in response to a HelloRequest message or whenever it wants to re-establish security parameters in an existing connection.

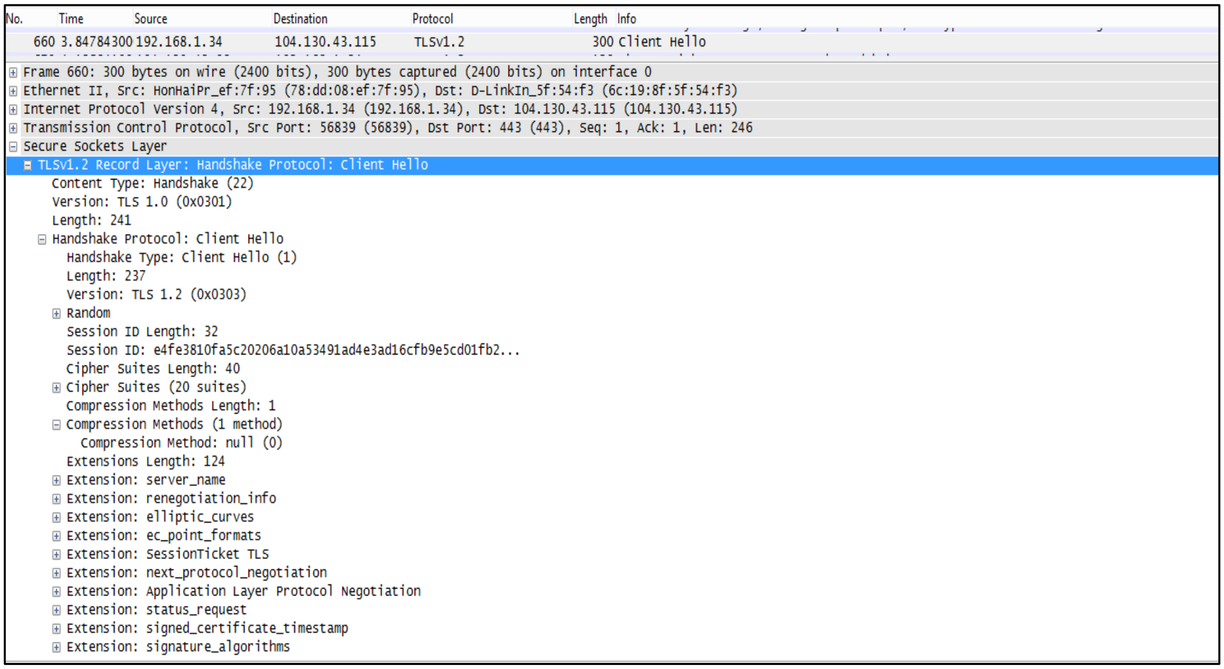
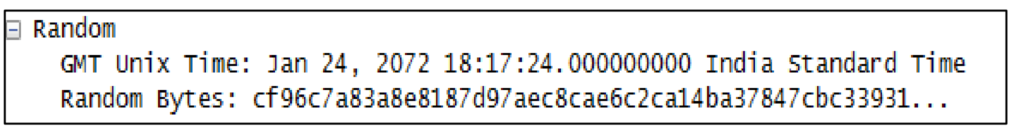
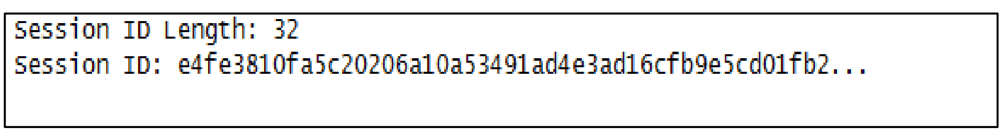


Figure 12: ClientHello

- **ProtocolVersion:** The version of TLS that a client wants to use.
- **Random**: A random structure containing client's time and 28 bit random number generated by the client.



- **SessionID**: The ClientHello message contains a session identifier which is null for a new connection. Session identifier can be same as that of an earlier connection or an existing connection.



- **CipherSuite**: It contains the list of algorithms supported by the client arranged in client preference. Each cipher suite is a combination of key exchange algorithm, bulk encryption algorithm, a MAC, and a PRN.

```
⊟ Cipher Suites (20 suites)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc14)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc13)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
    Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
    Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
    Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
    Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
```

- **CompressionMethod**: It lists the client supported compression methods sorted by client preference.

```
⊟ Compression Methods (1 method)
    Compression Method: null (0)
```

- **Extensions**: The client may request additional functionality from the server by using the extension field.

```
⊞ Extension: server_name
⊞ Extension: renegotiation_info
⊞ Extension: elliptic_curves
⊞ Extension: ec_point_formats
⊞ Extension: SessionTicket TLS
⊞ Extension: next_protocol_negotiation
⊞ Extension: Application Layer Protocol Negotiation
⊞ Extension: status_request
⊞ Extension: signed_certificate_timestamp
⊞ Extension: signature_algorithms
```

- **ServerHello -** Upon receiving ClientHello message, the server selects appropriate set of algorithms (protocol, cipher suite, and compression method) and responds with a ServerHello message. The structure of ServerHello message is similar to ClientHello message.
  - **ProtocolVersion:** The server will send the version of TLS the client wants to use if it supports that version or it will send an older version of TLS
  - **Random**: A random structure containing server's time and 28 bit random number generated independently by the server.
  - **SessionID**: On receiving a ClientHello message with non-null SessionID, the server checks its session cache. If a match for the SessionID is found, the server may resume the same using the previously established credentials session or may start a new session. The server may also return a null SessionID to indicate the client that the session will not be cached and hence cannot be resumed.
  - **CipherSuite**: It contains a single cipher suite selected by the server from the list of cipher suites sent by the client in the ClientHello message.
  - **CompressionMethod**: It contains a single compression method selected by the server from the list of compression methods sent by the client in the ClientHello message.

```
⊟ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 84
  ⊟ Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 80
      Version: TLS 1.2 (0x0303)
    ⊟ Random
        GMT Unix Time: Oct 15, 2014 15:08:09.000000000 India Standard Time
        Random Bytes: 7d1b774db899621e0da0dd3dfb660cd5017f9dd9d4b78986...
      Session ID Length: 0
      Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
      Compression Method: null (0)
      Extensions Length: 40
    ⊞ Extension: server_name
    ⊞ Extension: renegotiation_info
    ⊞ Extension: ec_point_formats
    ⊞ Extension: SessionTicket TLS
    ⊞ Extension: Unknown 30032
    ⊞ Extension: Application Layer Protocol Negotiation
```

Figure 13: ServerHello message

- **Certificate** - If the agreed upon key exchange algorithm uses certificates, the server immediately sends a server certificate message to the client. This message contains the server's certificate chain.

```
No.   Time        Source          Destination    Protocol    Length  Info
675   4.14805700  104.130.43.115  192.168.1.34   TLSv1.2     1414    Certificate

⊞ Frame 675: 1414 bytes on wire (11312 bits), 1414 bytes captured (11312 bits) on interface 0
⊞ Ethernet II, Src: D-LinkIn_5f:54:f3 (6c:19:8f:5f:54:f3), Dst: HonHaiPr_ef:7f:95 (78:dd:08:ef:7f:95)
⊞ Internet Protocol Version 4, Src: 104.130.43.115 (104.130.43.115), Dst: 192.168.1.34 (192.168.1.34)
⊞ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 56839 (56839), Seq: 1361, Ack: 247, Len: 1360
⊞ [2 Reassembled TCP Segments (2329 bytes): #674(1270), #675(1059)]
⊟ Secure Sockets Layer
  ⊟ TLSv1.2 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 2324
    ⊟ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 2320
        Certificates Length: 2317
      ⊟ Certificates (2317 bytes)
          Certificate Length: 1326
        ⊞ Certificate (id-at-commonName=s809.hoverzoom.net,id-at-organizationalUnitName=Domain Control Validated - RapidSS,id-at-organizationalUnitName=See
          Certificate Length: 985
        ⊞ Certificate (id-at-commonName=RapidSSL CA,id-at-organizationName=GeoTrust, Inc.,id-at-countryName=US)
```

```
⊟ Certificates (3655 bytes)
    Certificate Length: 1737
  ⊞ Certificate (id-at-commonName=*.google.com,id-at-organizationName=Google Inc,id-at-localityName=Mountain View,id-at-stateOrProvinceName=California
    Certificate Length: 1012
  ⊞ Certificate (id-at-commonName=Google Internet Authority G2,id-at-organizationName=Google Inc,id-at-countryName=US)
    Certificate Length: 897
  ⊞ Certificate (id-at-commonName=GeoTrust Global CA,id-at-organizationName=GeoTrust Inc.,id-at-countryName=US)
```

Figure 14: Certificate details

- **ServerKeyExchange** - This message is only sent by the server. It conveys cryptographic information to the client needed for communicating pre-master secret.

```
⊟ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 147
  ⊟ Handshake Protocol: Server Key Exchange
      Handshake Type: Server Key Exchange (12)
      Length: 143
```

- **CertificateRequest -** Depending on the negotiated cipher suite, a non-anonymous server can request for the client certificate form the client. This message, if sent, will immediately follow the ServerKeyExchange message.

- **ServerHelloDone -** This message is sent by the server to indicate to the client that it is done with its part of key exchange and the client can now proceed with its part of the key exchange. On receipt of this message the client should verify the server certificate and make sure that the parameters sent in server hello message are valid and acceptable.

```
⊟ TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 4
  ⊟ Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```

- **Client Certificate:** This is the first message sent by the client on receipt of the ServerHelloDone message. This message is sent only if certificate is requested by the client.

- **ClientKeyExchange:** This message is sent immediately after the client certificate (if it is sent) or immediately after the ServerHelloDone message. This message sets the premaster secret either by using RSA or Diffie-Hellman mechanisms. At the end of this message both client and server share the same premaster secret.

```
⊟ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 70
  ⊟ Handshake Protocol: Client Key Exchange
      Handshake Type: Client Key Exchange (16)
      Length: 66
```

2. **Change Cipher Spec Protocol**

The ChangeCipherSpec message is sent by both the client and the server to inform each other that for further communications the negotiated CipherSpec and keys will be used.

```
⊟ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.2 (0x0303)
    Length: 1
    Change Cipher Spec Message
```

3. **Application Data Protocol**

The record layer fragments, compresses, and encrypts the application data based upon the state of the connection.

```
⊟ TLSv1.2 Record Layer: Application Data Protocol: spdy
    Content Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 103
    Encrypted Application Data: 768a92c6510bf9f67bda6f460e94ff89e3b71c964a016d0a...
```

4. **Alert Protocol**

Alert messages convey the severity of the message along with its description. There are two types of alert messages; warning and fatal. Alert messages are used for error handling and closing the connection gracefully or terminating the connection in case of a fatal error.

A note on other encrypted services and Protocols

A few other protocols and services that use encryption are described below. These protocols are often referred to while talking about SSL/TLS in general.

IPSec – IPSec (Internet Protocol Security) provides encryption at the IP layer (Network layer of the OSI model). While SSL/TLS services work at the Presentation layer, IPSec specifically operates at the IP layer of the protocol suite. Thus when used, it provides encryption services for all protocols operating at the TCP and application layer. A typical example of usage for IPsec is the VPN tunnel which provides end-to-end encryption for all the network communication between two hosts. So what makes it different from SSL/TLS on a broad level? Unlike SSL/TLS, IPSec operates at the kernel level and cannot be implemented within the application code boundary and hence the application does not have any control over its security parameters.

SSL/TLS allows sufficient flexibility to the client-server model applications to have a finer control of their own security.

DTLS – DTLS or the Datagram Transport Layer Security is a protocol similar to SSL/TLS but operating over datagram packets, also known as the UDP traffic. Since UDP is a stateless protocol and unlike TCP is not a reliable mode of communication, slight and minimal changes are made in TLS protocol so that a majority of its features can be reused. The resultant protocol is named DTLS and is defined in RFC6347[11]. An example of software is by net-snmp package which uses UDP for sending its traps. Net-snmp can be configured to utilize DTLS for encrypting its UDP traffic.

WEP/WPA/Wireless Security - WEP and WPA protocols are meant to provide security over wireless networks. The wireless protocols encapsulate the entire application and transport layer traffic and typically form an encrypted tunnel between the client computer/device and the WiFi-access point. The security mechanisms are defined in IEEE 802.1x standards. Parts of wireless security can be considered similar to SSL/TLS mechanisms of authentication. However, wireless security is a far more complex subject due to the added complications of wireless technologies, hardware limitations of router devices, and greater vulnerability to eavesdropping as compared to wired technologies.

SSH – SSH or secure shell is an application level security mechanism which is primarily meant to secure the communication while connecting to Unix/Linux shells. Before the use of SSH, remote terminal connection protocols like Telnet and FTP used to transmit the entire content, including the passwords, in plaintext. The SSH protocol provides a way to authenticate the client and server by the user of public and private keys. SSH also supports tunneling and can be configured to forward existing port specific non-encrypted communication.

SMTPS/LDAPS/POP3S, etc.

There are a number of other traditional application layer protocols just like HTTP that do not have a built-in encryption mechanism, but can effectively use SSL/TLS encryption to secure their communication. So when HTTP is over SSL, it is named as HTTPS. Some examples include: SMTP (Simple Mail Transfer Protocol - used to send emails) which can use SMTPS for encrypted channel, POP3/POP3S (Post Office Protocol 3 - used to receive emails), IMAP/IMAPS (Internet Message Access Protocol), LDAP/LDAPS (Lightweight Directory Access Protocol), etc. Typically, these protocols use a standard port (i.e. port 25 for SMTP) for their communication and a different port for an SSL/TLS enabled counterpart (i.e. port 465 for SMTPS). However, there is another way to enable SSL/TLS communication on an existing non-SSL port. StartTLS is a way to convert an existing non-encrypted, insecure communication into an SSL/TLS secured one. The client can connect normally to a server and communicate in plaintext until it decides to upgrade the connection to an SSL one by issuing the STARTTLS command.

Difference between SSH and SSL

There is often confusion between the SSH and SSL service when we talk about secure network communication. The confusion is at its prime when vulnerabilities are discovered in SSL (or let's say an SSL library like OpenSSL) and IT folks are wondering if the vulnerabilities also affect SSH. A more prominent question asked by beginners these days is that if SSH and SSL are similar in capabilities, then why do we have two protocols? Now that SSL/TLS is so ubiquitous and its layers come as an extension of almost every other application, it could be hard to understand the real purpose it was developed for. It is best to understand the differences when you go through the history and the purpose of development for these protocols and the evolutionary part of computers during the 90s.

Historically, it would be very easy to understand the differences between SSH and SSL because they both were born to provide solutions to different problems.

In simple terms, SSH or the Secure Shell is an application layer protocol aimed at providing encrypted network communication for the traditional 'not-so-secure' telnet which opens a remote shell for Unix/Linux systems. All the older services like telnet, ftp, rcp from Unix days never used to encrypt data and it is trivial to sniff plaintext passwords and transferred data through these protocols. In earlier days, before the beginning of World Wide Web and web browsers, programs like telnet and ftp were extremely popular. While SSH is aimed at securing telnet and other Unix services like ftp, rcp, etc., SSL was more focused on securing the web-based communication in the advent of e-commerce possibilities. The efforts towards the development of SSL were initiated by Netscape which was pioneering browser development in the 90s.

SSH also supports tunneling or port forwarding where incoming data on a port is encrypted and forwarded by the SSH server. SSH and SSL do not have a direct relationship as far as their working mechanism is concerned. However, since both provide cryptographic services, some conceptual aspects of their operations may appear similar. For example, SSL/TLS connections require a trusted certificate, whereas SSH connections present a public fingerprint of the machine you are connecting to. Both of them intend to provide a means to the user to identify and trust the service they are connecting to.

Tunneling is another feature where SSL/TLS and SSH are conceptually similar, however the implementation varies. While acting as a tunnel, SSL/TLS does not care what application lies underneath; it could be HTTP, SMTP, LDAP, or anything. SSH has a tunneling capability where it can forward incoming data on a port without worrying about what it carries.

SSL/TLS connections do not always require client authentication, but SSH session requires the client to be authenticated via a password, key, or GSSAPI methods. Similarly, some of the hardening procedures for SSL/TLS and SSH can appear similar. Choice of ciphers enabled in SSH also matter in the same way it matters in SSL/TLS. Disabling of obsolete and insecure protocols like SSHv1 (like SSLv2 in SSL/TLS) also holds true for SSH.

Understanding Cipher-suites

A typical SSL session consists of a number of procedures to ensure confidentiality, integrity, and authentication in communication. Any book on basic public key cryptography explains these processes in detail. These processes require different kinds of algorithms and ciphers. A cipher-suite is a collection of ciphers that are collectively used in an SSL session. Each cipher in a cipher-suite serves a different purpose. The purpose includes the method used for key exchange and authentication, encryption algorithm, and MAC calculation (hashing) algorithm.

Let us break a typical cipher suite into its parts as defined in RFC 5246. A full-fledged cipher-suite name contains the following format:

```
TLS_KX_WITH_CIPHER_MAC
```

The first part is the protocol which is SSL or TLS. The second part - KX is - meant for key exchange which belongs to cipher algorithms that provide the key exchange feature, and authentication (if supported) like RSA, Diffie-Hellman (DH), etc. The cipher indicates the symmetric key algorithm like AES along with its mode of operation like CBC and finally followed by the hashing algorithm like MD5 and SHA. An example is:

```
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
```

Where - the fields indicate DHE for Diffie-Hellman key-exchange, RSA for authentication, AES with a 128 bit key in CBC mode for symmetric encryption and SHA (SHA-1) for message hash. For a detailed understanding and associated reading, refer to RFC 5246.

Cipher categories

Some popularly known cipher types are discussed here. Some of these ciphers crop up in security scanners and security hardening guides as to why they should be disabled or enabled. It is important that one have a fair idea about the existence of these ciphers and terminology used by vendors, because these are often a cause of confusion. Examples include null, export grade, low, medium and high grade, and anonymous ciphers.

Null ciphers are considered weak because they do not provide any real encryption. It does not mean that they are not important. These can only be used for testing and troubleshooting purposes or when confidentiality of the message is not needed.

`Examples`[12]`:`

```
SSL_RSA_WITH_NULL_MD5                   NULL-MD5
SSL_RSA_WITH_NULL_SHA                   NULL-SHA
```

Here RSA is used for key exchange, MD5 and SHA are used for Mac and Null cipher is used for ‘encryption’. The user can refer to the RFCs as suggested reading for a better understanding: RFC2410[13] and RFC4785[14].

Export ciphers - Export ciphers are intentionally weak for historic reasons and are not used in common deployments. Their encryption can be easily broken with basic hardware. (RFC2246) The security scanners check if export ciphers are enabled on your SSL deployment by mistake or otherwise. The typical key size is around 40-56 bits. By modern standards, any key size less than 128 bits is considered weak.

Anonymous cipher-suites – The anonymous cipher-suites do not provide a way of signature-based authentication in the SSL session and do not use a certificate. Hence, these are vulnerable to a man-in-the-middle attack. The implementation is expected to provide authentication by other means, for example a pre-shared secret key or password. For a better understanding see RFC4492[15].

Low, Medium, and High grade ciphers – Typically, this classification refers specifically to the symmetric encryption key size being used in an algorithm. Due to advancements in CPU and computing power, modern day cryptography considers any key less than 128 bit as weak. The Openssl's page[16], defines low-grade ciphers with 64 and 56 bit encryption algorithms. Medium refers to 128 bit and High implies encryption algorithms with key sizes greater than 128.

AEAD cipher-suites – The AEAD (Authenticated Encryption and Associated Data) is an advanced and relatively modern approach of block cipher mode of operation that is gaining popularity because of the weaknesses discovered in recent years in existing modes (e.g. CBC). Two of these modes are GCM (Galois Counter Mode) and CCM (Counter with CBC-MAC). RFC 5116, RFC5246, RFC5288, RFC5289, RFC5430 discuss these suites.

Forward secrecy – Forward secrecy and Perfect forward secrecy are properties of encrypted communications where the keys used for encrypting the session are random and do not rely on a single secret. This ensures that in the event of the single secret getting compromised, the other parts of encrypted communication cannot be compromised. This property is associated with Diffie-Hellman key change cipher-suites.

RC4 issues – The RC4 stream cipher used to be a popular choice because it was known to be immune to CBC mode attacks discovered in recent years. However, recently new attacks have been discovered against RC4 which are discussed later in the article (RC4 Biases).

Testing SSL/TLS for Security

Testing the SSL/TLS service for its strength and communication is an important part of verifying and qualifying your SSL/TLS communication.

There are a lot of tools available for SSL scanning and testing. Some tools are open-source, some are commercial, while some are available online. The reader is free to choose or evaluate the tools and scanners based on other requirements such as reporting, support, etc. This article serves an educational purpose and does not evaluate, recommend, or make a preference for any specific tool. For the purpose of this article we take the examples of open source tools that are recommended by OWASP[17].

Referring to the OWASP page for SSL gives examples of a lot of tools that can be used for testing SSL services. So where is the problem? Different tools test for different problems. A lot of tools do not get updated and are not maintained so they may not refer to the most recent problems associated with SSL/TLS. Some vulnerabilities are often disputed and are termed as features by vendors. An example of this is the client-side renegotiation feature where the community is divided. While evaluating those tools you will realize that a number of tools test the most common things, whereas a few test for special kind of attacks and recent vulnerabilities. Another element is the discovery of new vulnerabilities with time, so when you see a new tool which supports 'CCS scanning' you might be puzzled as to what CCS scanning refers to. To answer that, a recent vulnerability related to ChangeCipherSpec Injection (CVE-2014-0224[18]) in OpenSSL resulted in CCS scanning in some of the tools. It is obvious that for an effective SSL testing strategy one should be well versed in the common terminologies used in SSL/TLS, recent vulnerabilities, and the right/effective tools to be used in a particular situation or deployment.

The testing part can be divided primarily into the following sections:

- Setting the expectation right: whitelisting/backlisting of acceptable items, choice of standards and appropriate choice of SSL provider/library that supports your requirements: A theoretical study.
- Configuration inspection, correctly setting the configuration.
- Protections of secrets, key file permissions, etc.
- Testing for certificate issues Name mismatch, signing algorithm, public key strength, public key algorithm, key exchange mechanism, etc.
- Testing for protocols (SSLv2, SSLv3, TLS1, TLS1.1, TLS1.2)
- Protocol level issues (renegotiation, compression at SSL and at application level, such as HTTP)
- Supported Cipher-suites, ciphers, key sizes, etc.
- Specific vulnerabilities (Heartbleed, CRIME, Lucky13, etc.)

SSL issues and problems reported cannot be categorized as black and white. For some things, you may have to make a judgment call or decide upon an action plan for the future of your product. Suppose a security scanner identifies ciphers running in CBC mode and hence declares your setup vulnerable to the BEAST attack. Depending on the scanner, the suggested solution would be to use RC4. At the same time, another scanner reports vulnerabilities because RC4 is not considered secure. Probably every cipher-suite has some weakness or another. What does this mean? Is the whole SSL world going to crumble now? The conflicting reports often raise suspicion whether the problem is real or just exaggerated paranoia. We understand the possibility of an attack, but is it for real? Is it practical to conduct an attack without compromising additional infrastructure?

The truth is that there are external dependency factors that sometimes you must count in before the vulnerability can be exploited. An excellent published study/survey[19] on attacks from 2013 here elaborates on various SSL attacks, the dependencies, and practical feasibility of attacks. However, it must be remembered that with passing time and technology, advancements change and what is considered as theoretical can turn into real in the future.

Now, returning to testing of SSL service, let’s take an example of scanning an nginx web server in its default SSL configuration:

We use the TestSSLServer[17] tool which is a Java-based tool listed in the OWASP page for SSL/TLS testing:

```
root@kali:~# java -jar TestSSLServer.jar 192.168.32.146 443
Supported versions: SSLv3 TLSv1.0 TLSv1.1 TLSv1.2
Deflate compression: no
Supported cipher suites (ORDER IS NOT SIGNIFICANT):
  SSLv3
     RSA_WITH_3DES_EDE_CBC_SHA
     DHE_RSA_WITH_3DES_EDE_CBC_SHA
     RSA_WITH_AES_128_CBC_SHA
     DHE_RSA_WITH_AES_128_CBC_SHA
     RSA_WITH_AES_256_CBC_SHA
     DHE_RSA_WITH_AES_256_CBC_SHA
     RSA_WITH_CAMELLIA_128_CBC_SHA
     DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
     RSA_WITH_CAMELLIA_256_CBC_SHA
     DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
     TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
     TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
     TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  (TLSv1.0: idem)
  (TLSv1.1: idem)
  TLSv1.2
     RSA_WITH_3DES_EDE_CBC_SHA
     DHE_RSA_WITH_3DES_EDE_CBC_SHA
     RSA_WITH_AES_128_CBC_SHA
     DHE_RSA_WITH_AES_128_CBC_SHA
     RSA_WITH_AES_256_CBC_SHA
     DHE_RSA_WITH_AES_256_CBC_SHA
     RSA_WITH_AES_128_CBC_SHA256
     RSA_WITH_AES_256_CBC_SHA256
     RSA_WITH_CAMELLIA_128_CBC_SHA
     DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
     DHE_RSA_WITH_AES_128_CBC_SHA256
     DHE_RSA_WITH_AES_256_CBC_SHA256
     RSA_WITH_CAMELLIA_256_CBC_SHA
     DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
     TLS_RSA_WITH_AES_128_GCM_SHA256
     TLS_RSA_WITH_AES_256_GCM_SHA384
     TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
     TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
     TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
     TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
     TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
     TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
     TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
     TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
     TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
----------------------
Server certificate(s):
  2cb353398af1d75284213db6b0a825839de53c0c: CN=nginx-dev, OU=Nginx dev, O=Test Nginx,
L=Brisbane, ST=Queensland, C=AU
----------------------
Minimal encryption strength:     strong encryption (96-bit or more)
Achievable encryption strength:  strong encryption (96-bit or more)
BEAST status: vulnerable
CRIME status: protected
```

The output of the tool lists the protocol versions that the web server supports. It also lists what cipher-suites are supported at different protocol level like SSLv3, TLS 1.0, etc. It also tells you whether the SSL/TLS is protected or vulnerable to attacks like BEAST and CRIME. As it turns out, this web server's SSL/TLS can be vulnerable to BEAST attack. BEAST is more of an attack towards the clients. We may have to investigate what conditions make TestSSLServer tool report a website vulnerable to BEAST. The home page[20] for TestSSLServer notes the situations for its BEAST attack; in this case it is the presence of CBC mode ciphers. As mentioned in the Popular and Common Attacks section, CBC mode ciphers are vulnerable to BEAST attack.

Example 2: Scanning the server with nmap's ssl enumeration scripts[17]:

Nmap is a famous network port scanner and it results in a similar output with the protocols supported along with the ciphers.

Example 3: Another good option is to use sslyze[17] which in addition to listing protocols and supported cipher-suits, helps in finding out if client initiated renegotiations are supported or not.

Example 4: You may even use the Openssl client[17] to directly connect to the SSL server.

As long as a service supports the standard handshake procedure, these tools can be used to scan these services and check for any loopholes. There are services like SMTP which can have the provision of using STARTLLS for initiating the secure communication. For scanning such service, you may need a scanner that has STARTTLS support built in, e.g. sslyze supports starttls option for SMTP and xmpp services.

When scanning for services other than HTTPS, you may have to research the tools to find what options are supported.

```
root@kali:~# sslyze --starttls=smtp --tlsv1 test.smtp.com:587
 REGISTERING AVAILABLE PLUGINS
 -----------------------------
  PluginCompression
  PluginOpenSSLCipherSuites
  PluginSessionRenegotiation
  PluginSessionResumption
  PluginCertInfo
 CHECKING HOST(S) AVAILABILITY
 -----------------------------

   test.smtp.com:587                  => 192.168.32.146:587

 SCAN RESULTS FOR TEST.SMTP.COM:587 - 192.168.32.146:587
 --------------------------------------------------------

  * TLSV1 Cipher Suites :

      Preferred Cipher Suite:
        ECDHE-RSA-RC4-SHA        128 bits      250 2.0.0 OK ur2sm9690289pbc.51 - gsmtp

      Accepted Cipher Suite(s):
        RC4-MD5                  128 bits      250 2.0.0 OK z2sm9732910pdc.95 - gsmtp
        RC4-SHA                  128 bits      250 2.0.0 OK x16sm9677018pbt.70 - gsmtp
        AES256-SHA               256 bits      250 2.0.0 OK vl1sm9682220pbc.62 - gsmtp
        AES128-SHA               128 bits      250 2.0.0 OK rv6sm9864658pab.9 - gsmtp
        ECDHE-RSA-AES256-SHA     256 bits      250 2.0.0 OK l13sm9694193pbq.40 - gsmtp
        DES-CBC3-SHA             168 bits      250 2.0.0 OK jt8sm9713909pbc.6 - gsmtp
        ECDHE-RSA-AES128-SHA     128 bits      250 2.0.0 OK j8sm2476648pdo.79 - gsmtp
        ECDHE-RSA-RC4-SHA        128 bits      250 2.0.0 OK bn13sm9784876pdb.4 - gsmtp

      Rejected Cipher Suite(s):
        PSK-RC4-SHA                     TLS Alert - No ciphers available
        PSK-AES256-CBC-SHA              TLS Alert - No ciphers available
        PSK-AES128-CBC-SHA              TLS Alert - No ciphers available
…..
```

Sslscan[17] is another tool which can help you in scanning starttls services:

```
root@kali:~# sslscan --starttls --tlsv1 test.smtp.com:587
                   _
           ___ ___| |___  ___ __ _ _ __
          / __/ __| / __|/ __/ _` | '_ \
          \__ \__ \ \__ \ (_| (_| | | | |
          |___/___/_|___/\___\__,_|_| |_|

                  Version 1.8.2
             http://www.titania.co.uk
        Copyright Ian Ventura-Whiting 2009

Testing SSL server test.smtp.com on port 587

  Supported Server Cipher(s):
    Failed    TLSv1  256 bits  ECDHE-RSA-AES256-GCM-SHA384
    Failed    TLSv1  256 bits  ECDHE-ECDSA-AES256-GCM-SHA384
    Failed    TLSv1  256 bits  ECDHE-RSA-AES256-SHA384
    Failed    TLSv1  256 bits  ECDHE-ECDSA-AES256-SHA384
    Accepted  TLSv1  256 bits  ECDHE-RSA-AES256-SHA
    Rejected  TLSv1  256 bits  ECDHE-ECDSA-AES256-SHA
    Rejected  TLSv1  256 bits  SRP-DSS-AES-256-CBC-SHA
    Rejected  TLSv1  256 bits  SRP-RSA-AES-256-CBC-SHA
    Failed    TLSv1  256 bits  DHE-DSS-AES256-GCM-SHA384
    Failed    TLSv1  256 bits  DHE-RSA-AES256-GCM-SHA384
    Failed    TLSv1  256 bits  DHE-RSA-AES256-SHA256
    Failed    TLSv1  256 bits  DHE-DSS-AES256-SHA256
    Rejected  TLSv1  256 bits  DHE-RSA-AES256-SHA
    Rejected  TLSv1  256 bits  DHE-DSS-AES256-SHA
    Rejected  TLSv1  256 bits  DHE-RSA-CAMELLIA256-SHA
    Rejected  TLSv1  256 bits  DHE-DSS-CAMELLIA256-SHA
    Rejected  TLSv1  256 bits  AECDH-AES256-SHA
    Rejected  TLSv1  256 bits  SRP-AES-256-CBC-SHA
    Failed    TLSv1  256 bits  ADH-AES256-GCM-SHA384
    Failed    TLSv1  256 bits  ADH-AES256-SHA256
    Rejected  TLSv1  256 bits  ADH-AES256-SHA
    Rejected  TLSv1  256 bits  ADH-CAMELLIA256-SHA
….
```

Testing databases

Unlike web servers, testing the database for SSL may not be straightforward all the time. Unlike web servers, databases may not follow the standard handshake procedure; hence, the handshake using a traditional SSL/TLS client like OpenSSL's s_client may fail. The only way to negotiate a successful handshake with a database is to use the database's supported client or its API (the JDBC connector in its SSL mode). Enabling SSL/TLS for database communications is not a major requirement for a majority of users because most times the database is installed on the same server and the database communication is not over the network. However, in deployments where high security is desired, it is best to enable database communication over SSL/TLS and to test it appropriately. The challenges in enabling SSL/TLS for a database are:

1. Documentation not as extensive as for SSL/TLS in the web/HTTP layer

2. Difficulty in testing as standard handshake may not be supported
3. Different databases implementing SSL/TLS differently, hence procedures to check may vary
4. Configuration issues, lack of clarity

The last part about the configuration issues is the most important and often neglected by users. A user may try to configure a certain set of parameters to enable SSL. However, in the absence of proper testing and validation, you cannot be sure if the database was ever configured for using SSL correctly. Also, you may not have standard tools or scanners to check if the content is encrypted.

A simple use-case: as a user you may opt to enable SSL/TLS for a network channel to secure database replication. The configuration may well be as documented by the database vendor, but for testing you really want to see it on the wire. Let's see how one can debug the SSL/TLS traffic when it's not known if the SSL/TLS is enabled with the server or not. Here is an example capture of a mysql client connecting to an SSL enabled server:

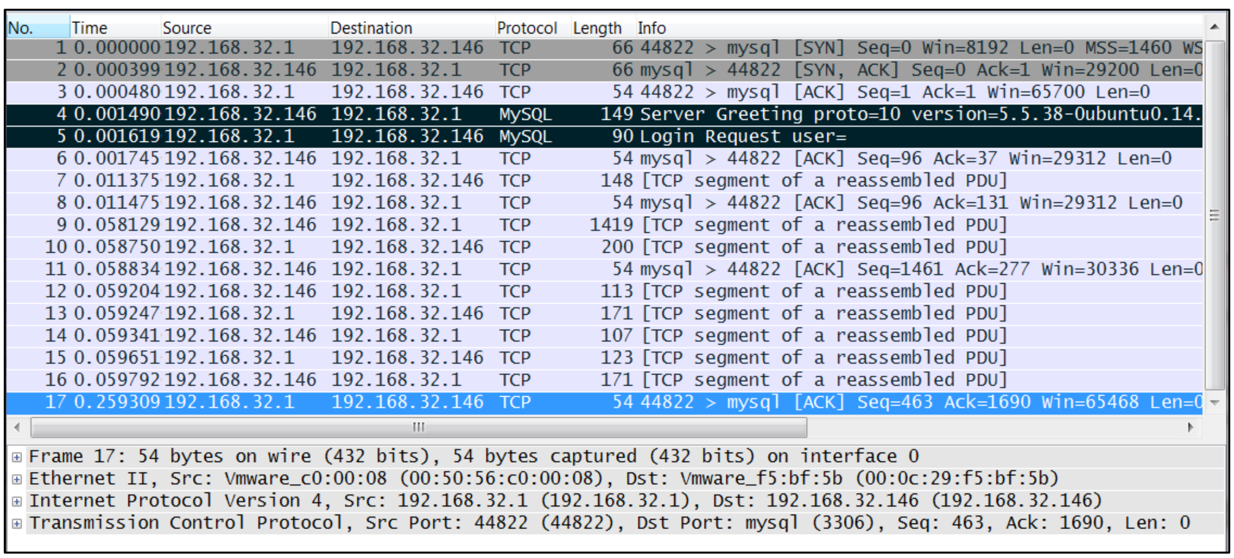


Figure 15: Mysql encrypted traffic - Before changing the decoding to SSL

The default capture without any filters would just display parts of traffic identified by Wireshark as MySQL traffic. The packets belonging to the MySQL protocol are marked in black for reference. You can even see the MySQL welcome string in the capture.

However, unlike a normal HTTPS communication, the SSL handshake does not occur after the 3-way TCP handshake. This is why typical SSL clients like openssl would fail while performing an SSL handshake with a MySQL database.

In order to see the TLS packets you can right click on any packet and select “Decode As” and then select SSL as the protocol. The next figure is what you see because now Wireshark has decoded all the packets as TLS packets and you can observe that our MySQL is using TLS version 1. Inspecting the packets would reveal all the information that MySQL’s TLS is using. The highlighted packets in yellow are the TLS handshake packets, while those in black were the original MySQL welcome string packets that don’t actually belong to TLS. Hence, Wireshark gives a message of “Ignored Unknown Record”.

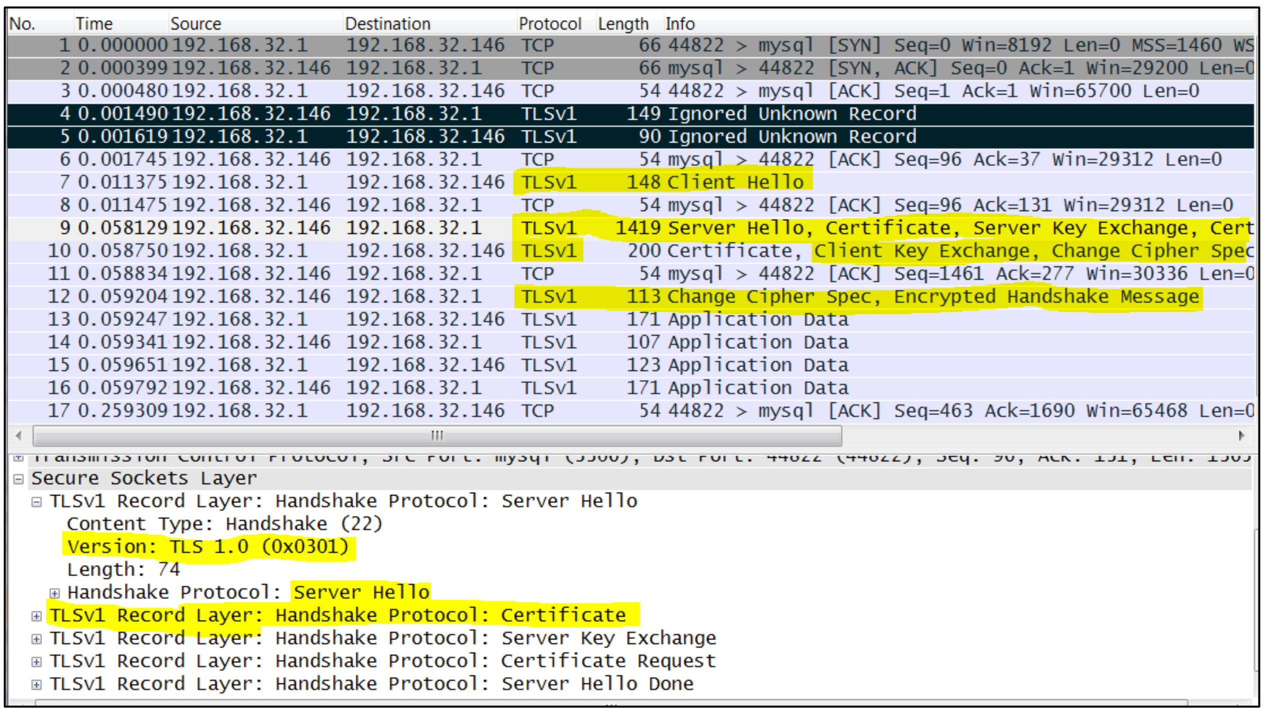


Figure 16: MySQL encrypted traffic - After changing the decoding to SSL

Looking at the Server Hello packet, you can see the cipher-suite selected by MySQL for communication in this session:

```
⊟ Secure Sockets Layer
  ⊟ TLSv1 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 74
    ⊟ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 70
        Version: TLS 1.0 (0x0301)
      ⊞ Random
        Session ID Length: 32
        Session ID: 88945bab4831fbcabdac6f06a6aba4f85139b9acd32e6869...
        Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
        Compression Method: null (0)
  ⊟ TLSv1 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 856
    ⊟ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 852
        Certificates Length: 849
      ⊞ Certificates (849 bytes)
  ⊟ TLSv1 Record Layer: Handshake Protocol: Server Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 397
```

Figure 17: Captured Cipher-suite in use for database traffic encryption

The above use case explains a scenario where the database SSL does not follow a standard SSL handshake and thus fails to be scanned by typical SSL scanners. The same concept of inspecting the handshake on wire can be extended for any other service that does not follow the standard handshake procedure.

SSL providers and Libraries

Talking about the SSL/TLS protocol and reading the RFCs for how it has been designed is one part of the learning. The RFCs are just a blueprint of how the SSL/TLS is supposed to work versus how SSL/TLS 'actually' works. The real strength lies in the implementation.

There are a number of cryptographic libraries that provide SSL/TLS implementation. The 'actual' working of protocols is dependent on how and to what extent they have been implemented in the software. Success of a new SSL/TLS blueprint depends on its adoption by the software vendors which is often slow due to the complexities involved. This blueprint can range from the introduction of a new protocol version, let's say like a new TLS 1.x specification, introduction of new cipher-suites or modes of block encryption (like the AEAD), introduction of new fields or specs, etc. Needless to say that the continuous evolving nature of web applications and the World Wide Web presents new challenges and risks to the SSL/TLS communication. The usefulness of a library lies in its continuous development and bug fixing, helpful documentation for the developers, ease of deployment and troubleshooting, and support for the most recent available protocols and cipher-suites. When we talk about 'usefulness', it strictly means your organization's requirements like security standards for operations involving cryptography, compliance levels, etc. An example is the NIST-approved cryptographic practices and the NSA Suit B cryptography defined in RFC6460[21], the FIPS 140 standard for the requirements of cryptographic modules, PCI compliance requirements, etc.

Knowledge of these libraries, what they are capable of, and where they are being used is essential in understanding the SSL/TLS implementation scene. A recent use case was the discovery of the Heartbleed bug that affected OpenSSL's implementation. Media reports[22] roughly quoted as 2/3 of the Internet affected by it. However, a number of users were clueless on where exactly OpenSSL is being used by the system and how to check if it is vulnerable.

The knowledge of where these libraries are being used also helps in the risk analysis of an SSL/TLS-related vulnerability. For instance, CVE-2014-0224[18] which is related to Man-In-The-Middle attack in OpenSSL, requires both client and server to be running a vulnerable version of OpenSSL. Does that mean that my browser is vulnerable to a Man-In-The-Middle attack when I am connecting to a website that uses OpenSSL on the server side? Probably, the answer is no. Because while the client is required to use OpenSSL, none of the popular browsers like Mozilla, Internet Explorer, and Google Chrome use OpenSSL; Mozilla and Chrome use NSS while IE uses SChannel respectively.

Talking about various libraries, different libraries have different licensing terms. While some are open source and free for general use, others are proprietary software. Among the most popular is the free and open sourced OpenSSL. Almost all popular Linux distributions come with a default installation of OpenSSL and it is easy to compile any server software in Linux, for example Apache web server or MySQL database with SSL support from OpenSSL. The open source and free nature has ensured that there is lot of help available on the Internet if you are stuck with an OpenSSL-related problem. Various projects like the LibreSSL have been forked out of OpenSSL's codebase. Then there are RSA's BSafe, Bouncy Castle, and JSSE (Java Secure Socket Extension) by Oracle. The JSSE implementation comes with JDK/JRE and Java-based programs typically use JSSE for their SSL implementation. An example is Oracle's

Weblogic Application server[23] which uses JSSE for its SSL implementation. The SSL configuration can be changed to use other providers as well. Apache Tomcat web application server[24] uses Java based JSSE and can alternatively use the native APR (Apache Portable Runtime) library which internally uses OpenSSL libraries. Microsoft's SChannel library is used by Microsoft products including its Internet Explorer browser. NSS (Network security services) is a library that is used by browsers like Mozilla and Google Chrome.

Checklist: Popular and common attacks in recent years

The following topic gives a brief introduction on the popular SSL/TLS vulnerabilities in recent years and complex technical details have been kept to a minimum. The readers are advised to follow references for a detailed understanding of the vulnerability in question. The discovery date highlights the approximate time of public disclosure by the researchers.

- **Vulnerability: Renegotiation vulnerabilities**
- **Discovery TimeLine: 2009-2011**
- **Checklist :**
- ✓ **Disable client initiated renegotiation**

CVE-2011-5094, CVE-2011-1473, CVE-2009-3555: Client-initiated renegotiation in an SSL communication can lead to the consumption of relatively more CPU cycles on the server side which can result in a Denial of Service situation on the server side. Though this vulnerability is disputed, where experts are divided on practicality of this attack, servers that allow renegotiation initiated by the client are considered vulnerable. Disabling client-initiated renegotiation is something that has to be supported and can be done from the server side SSL library.

- **Vulnerability: BEAST vulnerability**
- **Discovery Timeline: 2011**
- **Checklist:**
- ✓ **Use TLS 1.1 or TLS 1.2**
- ✓ **If not TLS 1.1 or TLS 1.2, disable CBC mode ciphers**
- ✓ **Use RC4**

The BEAST (Browser Exploit Against SSL/TLS) vulnerability (CVE-2011-3389) discovered in September 2011 is a vulnerability that affects SSL/TLS (versions 1.0 and earlier) and primarily works on the client side (through the web browsers). The vulnerability affects CBC mode ciphers. The server side mitigations include the preference of RC4-based cipher suites, while on the client (browser) side, it is recommended to use TLS 1.1 and TLS 1.2. The vulnerability is mostly reported by security scanners when CBC mode ciphers are in use. The popular mitigation is to disable CBC mode ciphers on the server side SSL configuration and use RC4. Note that use of RC4 was discouraged later, because of the discovery of RC4 Biases.

- **Vulnerability: CRIME vulnerability**
- **Discovery Timeline: 2012**
- **Checklist :**
- ✓ **Disable the use of HTTP compression**
- ✓ **Disable the use of compression in SSL/TLS**

The CRIME (Compression Ratio Info-leak Made Easy) attack (CVE-2012-4949) discovered in September 2012 works against compression techniques in use by web servers in an HTTPS connection. This vulnerability is reported by security scanners when compression is enabled in an SSL connection. The common solution is to keep compression in SSL/TLS disabled. The vulnerability is reported by the scanners when compression in SSL/TLS or even in HTTP is enabled.

- **Vulnerability: Lucky13 Attack**
- **Discovery Timeline: 2013**
- **Checklist :**
- ✓ **Disable the use of CBC mode ciphers**

Lucky 13 attack (CVE-2013-0169), discovered in February 2013, is a timing attack that aims to recover plain text when CBC mode ciphers are in use. Security scanners mostly report the vulnerability when CBC mode ciphers are in use.

- **Vulnerability: RC4 Biases**
- **Discovery Timeline: 2013**
- **Checklist :**
- ✓ **Disable the use of RC4 ciphers**

The RC4 algorithm has been a popular choice when attacks against CBC mode ciphers were on the rise. However, recent studies released in July 2013 indicated that RC4 has its own share of weaknesses (CVE-2013-2566). The mitigation is to move away from RC4. Security scanners report RC4 issues if RC4 is found as an enabled cipher in SSL/TLS.

- **Vulnerability: TIME vulnerability**
- **Discovery Timeline: 2013**
- **Checklist:**
- ✓ **Disable the use of HTTP compression**
- ✓ **Disable the use of SSL/TLS compression**

The TIME (Timing Info-leak Made Easy) vulnerability, discovered in April 2013, targets compression and mainly addresses the limitations of CRIME attack. The vulnerability is reported by the scanners when compression is enabled.

- **Vulnerability: BREACH vulnerability**
- **Discovery Timeline: 2013**
- **Checklist:**
- ✓ **Disable the use of HTTP compression**
- ✓ **Disable the use of SSL/TLS compression**

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (CVE-2013-3587), discovered in September 2013, is considered a successor to the CRIME vulnerability and targets the compression in the HTTP protocol itself in contrast to CRIME which targets the compression in SSL/TLS protocol. The vulnerability is reported by the scanners when compression is enabled.

- **Vulnerability: HeartBleed vulnerability**
- **Discovery Timeline: 2014**
- **Checklist:**
- ✓ **Upgrade your OpenSSL library**

The Heartbleed vulnerability (CVE-2014-0160), discovered in April 2014, was among the most serious bugs affecting SSL/TLS in recent times. The bug affected the OpenSSL package which is a famous and widely used implementation of SSL/TLS. The bug affected OpenSSL's extension named Heartbeat, hence the name HeartBleed.

- **Vulnerability: POODLE vulnerability**
- **Discovery Timeline: 2014**
- **Checklist:**
- ✓ **Disable the use of CBC mode ciphers**
- ✓ **Disable the use of SSLv3 in general (as only remaining cipher RC4 is considered insecure as well)**

The POODLE (Padding Oracle On Downgraded Legacy Encryption) vulnerability (CVE-2014-3566), disclosed in September 2014, affects SSL v3 and results in plain text discovery when CBC mode ciphers are used. A new variant of POODLE (CVE-2014-8730) has been announced in December 2014 that affects SSL implementations of certain vendors. Since the new variant is not a vulnerability in the protocol itself, a patch update from the vendor will fix the POODLE variant vulnerability.

Recommendations for selecting, configuring, and installing TLS server and clients

Testing SSL/TLS is a non-trivial and extensive task because of the large number of implementations, cipher suites, and exponential development in the field of crypto analysis. It is a research-oriented field and understanding its features in itself can take a long time. Now combine this time with the business requirement of, for example, online ecommerce and you have two conflicting areas to focus on. Different businesses have different security requirements. Systems in high security zones like governance, defense, and military may have to maintain the highest standards of security. A credit card payment processing business may have to worry about PCI compliance factors and everyday threats from the unruly Internet. A simple social networking platform with no sensitive financial data may only have to worry about usernames and passwords. Clearly, the risk is not the same everywhere.

A simple login-based application may use SSL/TLS just for the sake of it. They may not worry about the differences between TLS 1.2 and TLS 1.1, but the defense systems will certainly have to.

It is always wise to refer to a recommendation or standard when you are analyzing the security of your SSL/TLS system. The problem is at times these standards may become overwhelming and difficult to understand for common users.

Here are some recommendations suggested for SSL/TLS usage and testing:

NIST Guidelines for TLS implementations

NIST has defined a set of tests recognized as the minimum criteria for SSL/TLS testing. For those who want to achieve NIST-approved standards of SSL/TLS, this is the guide to look for. The guide is extensive in detail and covers recommendations for clients, servers, certificates, keys, and cipher related aspects of SSL.

http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf

OWASP Transport Layer Protection CheatSheet and Testing Guide

For those who do not want to go into details covered by NIST standards, a simpler option is the OWASP Transport Layer Protection cheatsheet which provides guidelines and a model to follow for protecting an application using TLS. It is written in a simple to understand language and can be an easy reference for anyone trying to build a secure application.

https://owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

OWASP has another guide for testing SSL/TLS services. This guide provides an easy way to scan your services with open source tools and can be a good source of guidance to create a test strategy for your SSL/TLS testing efforts.

https://owasp.org/index.php/Testing_for_SSL-TLS_(OWASP-CM-001)

Summary and Conclusion

The purpose of this article is not to repeat the same old cryptography basics and explain the Alice and Bob communication problem, but rather to delve into SSL in a practical way to solve the day-to-day problems and dilemmas engineers face. Its main idea is to equip the reader with knowledge and tools required to understand and troubleshoot on his/her own the typical SSL/TLS-related problems and impart the know-how to understand and evaluate risk for complex security issues that periodically crop up with scary names like HeartBleed, POODLE, TIME, and CRIME.

We started by exploring the basic history of SSL/TLS, its development, and major changes in successive protocol versions along with references to official RFCs one can always reference for a detailed study. Though SSL and TLS are often used interchangeably, TLS is the successor of SSL v3. The history timeline is followed by a 2010 SSLLabs report where we get an approximate idea on the adoption of various protocol versions of SSL/TLS. This presents us with an important insight that even though SSL/TLS has existed for nearly two decades, the adoption toward better and secure versions is still catching up.

We then discuss the three pillars of secure communication: **Confidentiality** provided by the encryption algorithm, **Integrity** by hashing algorithm, and **Authentication** by certificates after taking an example cipher-suite. We also explore the anatomy at packet level of an SSL/TLS network session. We briefly define what constitutes a SSL/TLS network packet and go through its fields, sub protocols used, and their relevance using Wireshark, followed by a structural explanation of the SSL/TLS sub protocols. In contrast to SSL/TLS, we briefly touch upon some of the other similar protocols like the IPSec, DTLS, WPA/WEP, SSH, etc., which often are a source of confusion to beginners. We discuss the conceptual differences between SSH and SSL. We go through the cipher-suite string of an SSL/TLS communication and explore various categories of ciphers that are important from a security perspective. Null, export, anonymous, low/high/medium grades, AEAD ciphers, and forward secrecy are terms you must know before you read an SSL security report. We also explore ways to scan a typical SSL service for the kind of ciphers and the SSL/TLS protocol version it supports using the OWASP recommended tools. Some scanners scan for specific vulnerabilities like renegotiation, CRIME, BEAST, etc. We also look for services like SMTP that use STARTTLS to initiate SSL while running their SMTP service on a standard port. Some services like the SSL on a database (MySQL) do not follow the typical SSL/TLS handshake procedures and an attempt to run a standard SSL client may fail during handshake; we elaborate on how to check at the packet level for such services. You may always go ahead and write your own scanner client with such info.

Talking about TLS versions is like going through the blueprints and is incomplete unless we have the knowledge of its implementation. Different software use different cryptographic libraries for their SSL/TLS use. To name a few, we talk briefly about various SSL libraries and providers like OpenSSL, NSS, SChannel, LibreSSL, RSA BSafe, Bouncy Castle, JSSE, and examples on where exactly they fit in and how they are used. We discuss use cases like CVE-2014-0224 on how this knowledge helps in analyzing SSL/TLS-related vulnerabilities and threats. The reader can choose to research more on these libraries and become familiar with the other implementations and figure out any limitations on their own. This certainly helps in choosing the appropriate library for one's long-term business use.

The next part explains the recent attacks discovered on SSL/TLS that have gained popularity over the past few years like Renegotiation problems, BEAST, CRIME, Lucky13, TIME, RC4 Biases, BEACH, HeartBleed, and POODLE. These are the vulnerabilities that you need to know and are commonly reported by SSL scanners while scanning an SSL service. Their associated CVE ID is mentioned for an easy reference. Often, different scanners suggest different mitigations and it creates more confusion for an average user on what exactly needs to done. Rather than repeating the complex technical details, this section attempts to provide clarity on these issues along with their timeline of disclosure and a checklist of what mitigations they actually call for along with a brief description. The reader becomes familiar with troubleshooting SSL/TLS-based services, which can result in rapid decision making and planned adaptation with the appropriate choice of SSL/TLS and understanding of prevalent attacks.

References

1. Netscape release of SSLv3 - https://web.archive.org/web/19970614020952/http://home.netscape.com/newsref/std/SSL.html
2. RFC for SSLv3 - http://tools.ietf.org/html/rfc6101
3. RFC for TLS v1.0 - http://tools.ietf.org/html/rfc2246
4. RFC for TLS v1.1 - http://tools.ietf.org/html/rfc4346
5. RFC for TLS v1.2 - http://tools.ietf.org/html/rfc5246
6. Updates for all protocols with SSL v2 - http://tools.ietf.org/html/rfc6176
7. SSLLabs report - http://blog.ivanristic.com/Qualys_SSL_Labs-State_of_SSL_2010-v1.6.pdf
8. X.509 v3 certificate - RFC5280 - https://www.ietf.org/rfc/rfc5280.txt
9. X.509 v3 certificate - RFC6818 - https://www.ietf.org/rfc/rfc6818.txt
10. Pre-shared key cipher-suites - RFC5487 - https://www.ietf.org/rfc/rfc5487.txt
11. DTLS Protocol - RFC6347 - https://www.ietf.org/rfc/rfc6347.txt
12. Openssl ciphers reference - https://www.openssl.org/docs/apps/ciphers.html
13. Null ciphers - RFC2410 - https://www.ietf.org/rfc/rfc2410.txt
14. Null ciphers - RFC4785 - https://www.ietf.org/rfc/rfc4785.txt
15. Anonymous ciphers - https://www.ietf.org/rfc/rfc4492.txt
16. Low, Med and High ciphers - https://www.openssl.org/docs/apps/ciphers.html#command_options
17. OWASP SSL testing tools – https://owasp.org/index.php/Testing_for_SSL-TLS_(OWASP-CM-001)
18. CVE-2014-0224 - https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0224
19. Survey on attacks - https://www.isecpartners.com/media/106031/ssl_attacks_survey.pdf
20. TestSSLServer home - www.bolet.org/testsslserver/
21. NSA Suit B cryptography - https://www.ietf.org/rfc/rfc6460.txt
22. Media report on HeartBleed - http://arstechnica.com/security/2014/04/critical-crypto-bug-in-openssl-opens-two-thirds-of-the-web-to-eavesdropping/
23. Weblogic Application server SSL configuration - http://docs.oracle.com/cd/E23943_01/web.1111/e13707/ssl.htm#SECMG384
24. Apache Tomcat SSL configuration - http://tomcat.apache.org/tomcat-8.0-doc/ssl-howto.html

General links for more information:

25. TIME attack BlackHat presentation - https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf
26. BREACH attack - https://media.blackhat.com/us-13/US-13-Prado-SSL-Gone-in-30-seconds-A-BREACH-beyond-CRIME-Slides.pdf
27. Lucky 13 attack - http://www.isg.rhul.ac.uk/tls/TLStiming.pdf
28. RC4 Biases attack - http://www.isg.rhul.ac.uk/tls/RC4biases.pdf