Understanding TLS

Tim Zielke

Written in 2019

Purpose of Presentation

- The main purpose of this Understanding TLS presentation is to give the reader a high level understanding of how TLS works.

- When doing administration or programming with TLS, it is helpful to have a working understanding of the TLS protocol. TLS is broken into two protocols. The handshake protocol and the record protocol. This session will mainly focus on the handshake part of the TLS protocol, but will also cover some of the record protocol, too.

- This presentation was originally designed to be given at an IBM MQ conference, so it has an IBM MQ slant. However, most of the content applies to TLS, in general.

- Presentation Note - The notes section for some slides are placed on the following slide. I felt this formatted better for reading them in a PDF.

Presentation References

- The following are references that were used to write this Understanding TLS presentation.
- TLS 1.2 RFC 5246 -> https://www.ietf.org/rfc/rfc5246.txt
- TLS 1.3 RFC 8446 -> https://tools.ietf.org/html/rfc8446
- “SSL and TLS - Designing and Building Secure Systems” book by Eric Rescorla
- IBM MQ manuals in the IBM Knowledge Center.
- Diagram References:

1) The TLS 1.2 handshake diagrams in this session were taken from the RFC 5246 document.
2) The TLS 1.3 handshake diagram in this session was taken from the RFC 8446 document.
3) The certificate chain diagram in this session was taken from the IBM MQ Knowledge Center.

What is TLS?

- TLS stands for Transport Layer Security
- TLS is a cryptographic protocol which enables two parties (client and server) to identify and authenticate each other and communicate with confidentiality and data integrity.
- TLS evolved from the Netscape SSL (Secure Socket Layer) v3 protocol, and is supported by the Internet Engineering Task Force (IETF). The phrase SSL is very pervasive in MQ security terminology (e.g. SSLCIPH, SSLPEER on channel definition), so it can be common now to see SSL used when it really just means TLS.
- Current versions of TLS that are recommended from a security perspective are TLS 1.2 and 1.3.

TLS - Asymmetric/Symmetric Encryption

- Before we look more at TLS, it is helpful to understand asymmetric and symmetric encryption.

- Asymmetric encryption - Plaintext is encrypted into ciphertext and then decrypted back to plaintext but with DIFFERENT keys (public and private) used in the encryption and decryption process.

  Example: RSA or Elliptic Curve

- Symmetric encryption - Plaintext is encrypted into ciphertext and then decrypted back to plaintext with the same key used in both the encryption and decryption process.

  Example: AES-256-CBC or RC4

Notes From Previous Slide

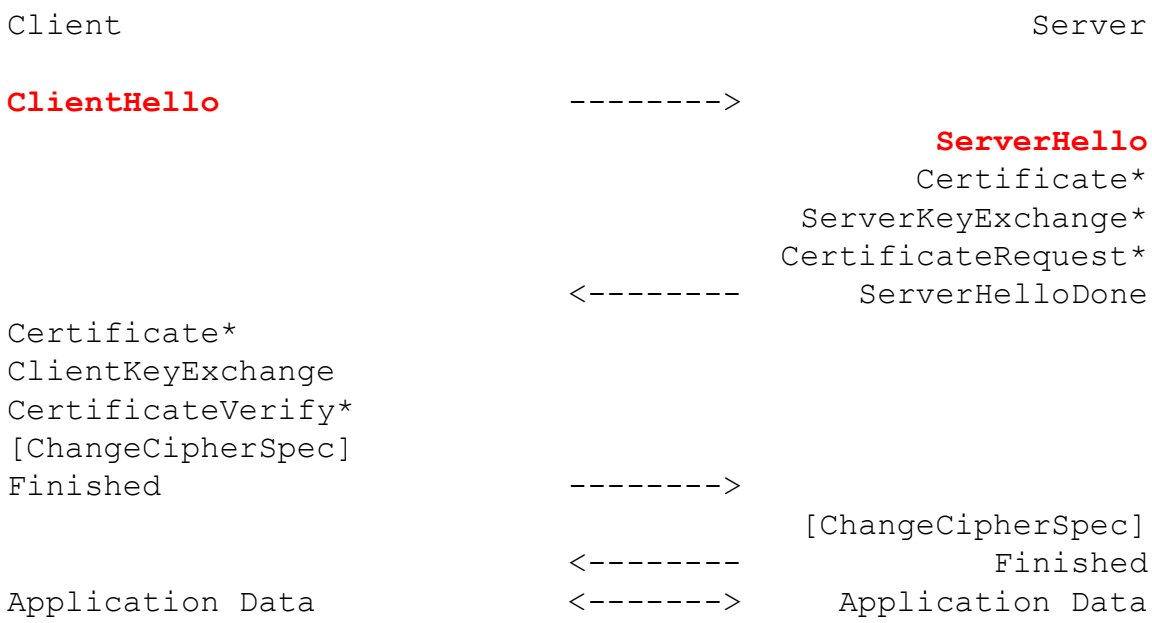
- For an example of asymmetric encryption, a TLS client end could use a public key to encrypt data with RSA. The TLS server end could then decrypt that encrypted data with the corresponding private key and RSA. Also, the TLS client end could use a private key to encrypt data with RSA. The TLS server end could then decrypt the encrypted data with the corresponding public key and RSA.

- For an example of symmetric encryption, the TLS client could encrypt data with a symmetric key and AES-256-CBC. The TLS server end could decrypt the data with the same symmetric key and AES-256-CBC.

TLS - Symmetric Encryption Is The Objective

- Generally speaking, the objective for TLS is for the client and server to be able to send encrypted data to each other using symmetric encryption (e.g. AES-256-CBC). Symmetric encryption is much faster than asymmetric encryption.
- In the next several slides, we will now look at the TLS 1.2 handshake and how it achieves that objective.
- Remember: “TLS is a cryptographic protocol which enables two parties (client and server) to identify and authenticate each other and communicate with confidentiality and data integrity.”
- We will also be looking at how TLS achieves identification, authentication, and communication with confidentiality and data integrity through the handshake and record protocol.

TLS 1.2 Handshake - Choose a CipherSuite



* Indicates optional or situation-dependent messages that are not always sent.

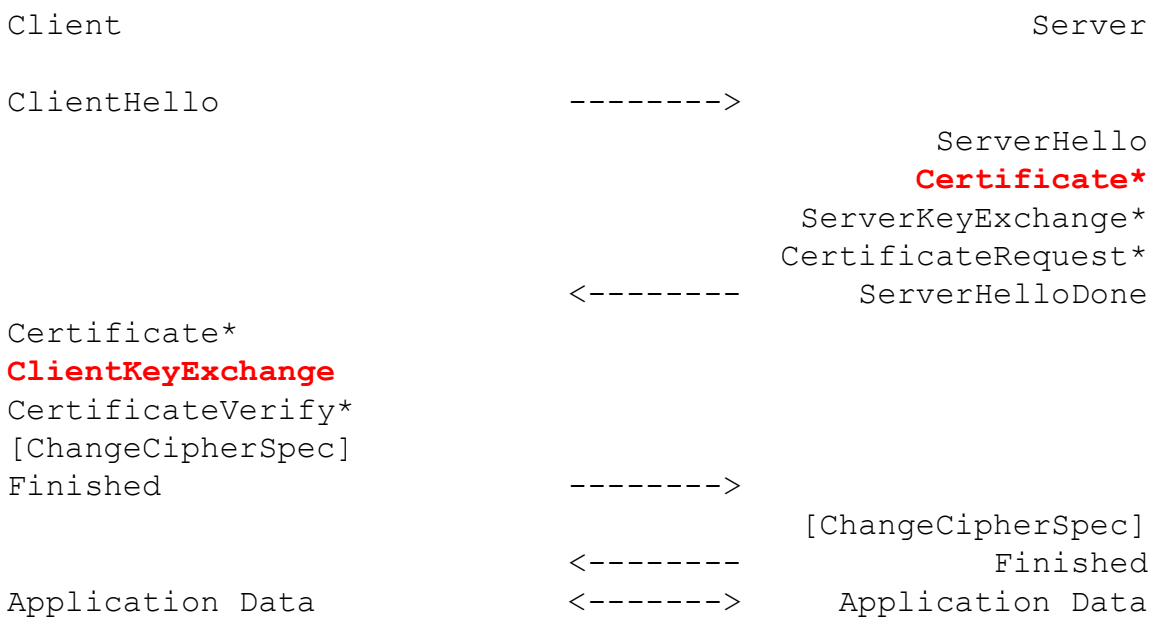
NOTES:

- ClientHello contains list of supported CipherSuites by the client.
- ServerHello contains the chosen CipherSuite. For our example, TLS_RSA_WITH_AES_128_CBC_SHA256 will be chosen.
- In the ClientHelllo and ServerHello, each side is also passing a random byte string to be used in subsequent computations of the symmetric keys.

Notes From Previous Slide

- Here we are looking at the TLS 1.2 handshake using a CipherSuite example of TLS_RSA_WITH_AES_128_CBC_SHA256.
- In this example, an MQ Java client called JavaClient will play the role of the TLS Client, and an IBM MQ queue manager called QM1 will play the role of the TLS Server.
- The JavaClient will use a Java keystore that has an RSA private key and public key certficate with a CN=JavaClient. The JavaClient certficate is signed by an Issuer of SubCA1. The keystore will contain two signer certificates. One signer certificate for SubCA1 with an Issuer of RootCA1. The other signer certificate for RootCA1.
- QM1 will use a key.kdb that has an RSA private key and public key certficate with a CN=QM1. The QM1 certficate is signed by an Issuer of SubCA1. The key.kdb will contain two signer certificates. One signer certificate for SubCA1 with an Issuer of RootCA1. The other signer certificate for RootCA1.
- In the handshake diagram above, the TLS Client and Server will use a ClientHello and ServerHello (bold red in diagram above) message to agree on a CipherSuite. In this first example that we look at, they will agree on the TLS_RSA_WITH_AES_128_CBC_SHA256 CipherSuite.
- The end objective of this TLS handshake example is for the TLS Client and Server to be able to send encrypted data to each other with the data being encrypted with symmetric encryption. In this example, they are agreeing to use AES_128_CBC for the symmetric encryption.

TLS 1.2 Handshake - Share a Secret



* Indicates optional or situation-dependent messages that are not always sent.

NOTES (for our example TLS_RSA_WITH_AES_128_CBC_SHA256):

- Server uses the Certificate message to send its RSA public key certificate to the client.
- Client uses the ClientKeyExchange message to send a premaster secret encrypted with the server's RSA public key.
- Server uses its RSA private key to decrypt the premaster secret.
- Both sides now have the previously sent random bytes and premaster secret which allows them to generate symmetric keys for AES_128_CBC symmetric encryption and a MAC (Message Authentication Code).

Notes From Previous Slide

- This is where the magic happens with the TLS 1.2 handshake. For our TLS_RSA_WITH_AES_128_CBC_SHA256 example, the handshake will use the TLS Server's RSA public/private keys to share an encrypted pre-master secret between the TLS Client (JavaClient) and Server (QM1). This pre-master secret is the secret piece that will allow the TLS Client and Server to be able to generate symmetric keys for the encryption and authentication of the data that will be sent between the TLS Client and Server when the handshake is completed.

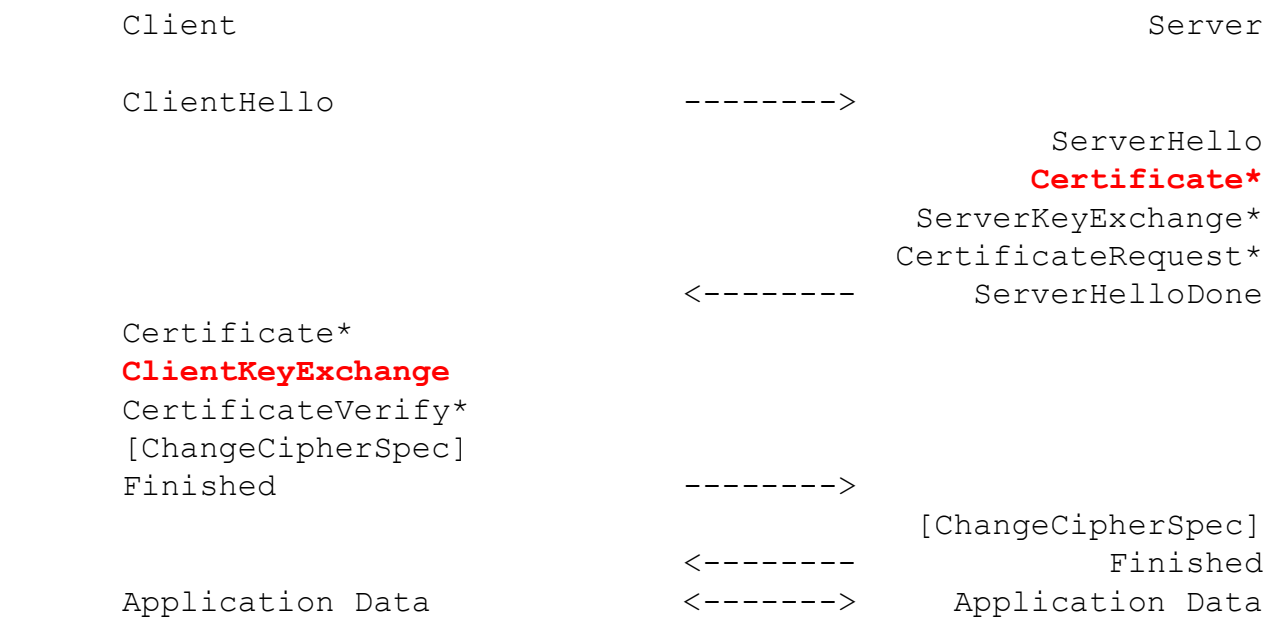
- In this example, QM1 will send its personal certificate to the JavaClient through the Certificate message (bold red in diagram abive). This QM1 personal certificate contains the RSA public key for QM1.

- The JavaClient will need to authenticate the QM1 certificate. We will talk about that more in the next few slides.

- Once the JavaClient has authenticated the QM1 certificate, it will send a pre-master secret to QM1. This pre-master secret will be encrypted using RSA and the public key of QM1 in the ClientKeyExchange message (bold red in diagram above). QM1 will then decrypt the pre-master secret using RSA and the private key from its key.kdb.

- We also mention a MAC (Message Authentication Code) in the above slide. There will be more on this in later slides, but the MAC is a way to authenticate the encrypted data records (sent between the TLS Client and Server after the handshake completes) and prevent the data records from being tampered with. The SHA256 piece of the CipherSuite is the HMAC-SHA256 algorithm that will be used to build this MAC on the encrypted data records.

TLS 1.2 Handshake - Authenticate The Server



* Indicates optional or situation-dependent messages that are not always sent.

NOTES (for our example TLS_RSA_WITH_AES_128_CBC_SHA256):

- Before the client can use the server's RSA public key, it needs to authenticate or trust the certificate.
- This authentication is done with a trust store that is on the client side.

Notes From Previous Slide

- The JavaClient does need to authenticate the QM1 certificate. We will see more details of how this authentication process happens in the next few slides.
- Remember we said the JavaClient has the following keystore:
- “The JavaClient will use a Java keystore that has an RSA private key and public key certficate with a CN=JavaClient. The JavaClient certficate is signed by an Issuer of SubCA1. The keystore will contain two signer certificates. One signer certificate for SubCA1 with an Issuer of RootCA1. The other signer certificate for RootCA1.”
- The SubCA1 and RootCA1 signer certificates in the JavaClient keystore will be critical in doing this authentication. As a note, you need to be careful that the signer certificates that you put in a keystore or key.kdb are ones that you trust, as they are used in the TLS handshake process to authenticate certificates from a remote end.

Digital Certificates

- A digital certificate protects against impersonation and certifies that a public key belongs to a specified entity. IBM MQ supports certificates that comply with the X.509 standard.

```
Subject : "CN=QM1,OU=MQ.DEV,L=OurTown,O=OurCompany,ST=Illinois,C=US"

Public Key :  30 82 01 22 30 0D 06 09 2A 86 48 86 F7 0D 01 01 (partially shown)
```

- NOTE: The private key is NOT included in the certificate.

- A certificate can be self-signed (Subject name matches the Issuer Name) or signed by a Certificate Authority (Ex. Verisign). A root CA certificate is a self-signed certificate.

```
Issuer : "SubCA1,CN=SubCA1,OU=IS Security,O=OurCompany ,L=OurTown,ST=Illinois,C=US"

Signature Algorithm : SHA256WithRSASignature
```

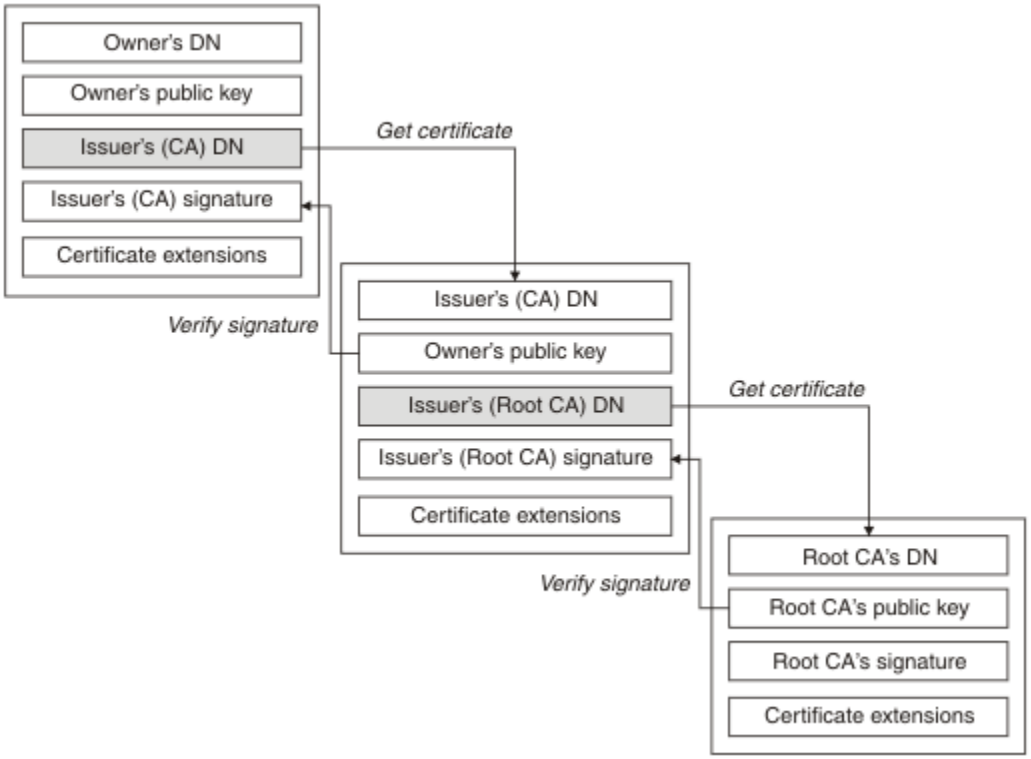
- When the certificate is for an individual entity, the certificate is called a *personal certificate* or *user certificate*.

- When the certificate is for a Certificate Authority, the certificate is called a *CA certificate* or *signer certificate*.

Notes From Previous Slide

- Here we look at some details of the QM1 certificate that will be sent to our JavaClient.
- The Subject portion of the certificate is where you have your distinguished name.
- The SSLPEER attribute on an MQ channel can be used to enforce that a received certificate's distinguished name must match certain criteria (e.g. CN=QM1). If you include SERIALNUMBER in your SSLPEER, it does provide a validation check that makes it very difficult for someone to try to create another copy of the certificate you are checking for.
- The Public Key portion is the RSA public key for QM1. Note that the private key is not included in the public certificate. The private key is stored securely on the remote end in the QM1 key.kdb file.
- If the Subject and Issuer had been identical, this would be a self-signed certificate. However, they do not match so this is CA signed certificate.

Certificate Chain



The certificate chain, also known as the certification path, is a list of signer CA certificates used to authenticate a personal certificate (e.g. queue manager certificate).

The chain, or path, begins with the personal certificate, and each certificate in the chain is signed by the entity identified by the next certificate in the chain. The chain terminates with a root CA signer certificate. The root CA signer certificate is always signed by the certificate authority (CA) itself (i.e. Owner DN and Issuer DN are the same). The signatures of all certificates in the chain must be verified until the root CA signer certificate is reached.

Notes From Previous Slide

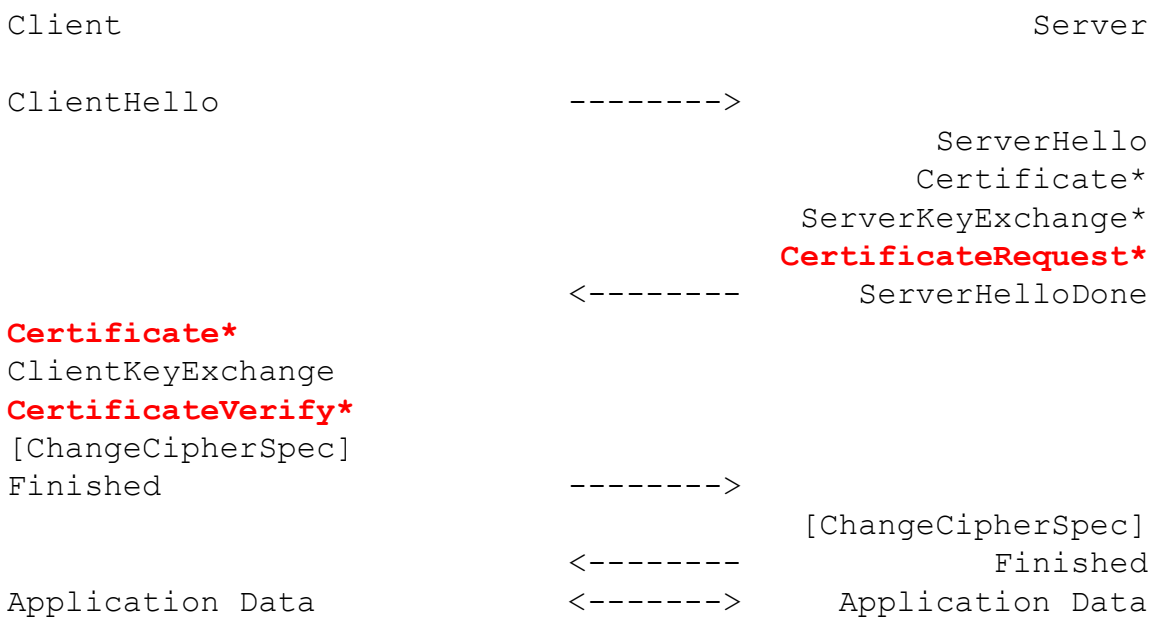
- Here we follow how the JavaClient will validate the QM1 certificate using the JavaClient's keystore.
- Remember – For the JavaClient keystore, it contains two signer certificates. One signer certificate for SubCA1 with an Issuer of RootCA1. The other signer certificate for RootCA1 with an Issuer of RootCA1.
- For the sake of simplicity, we will assume that the Issuer on the certs have shortened names like SubCA1 and RootCA1.
- In the top left box of the above diagram is the QM1 certificate. It has a Subject that starts with CN=QM1 and an Issuer of SubCA1, so it is a CA signed certificate since the Subject and Issuer do not match.
- The JavaClient will then look in its keystore for a signer certificate with a Subject of SubCA1, which it finds. The JavaClient then takes the public key from the SubCA1 signer cert and uses it to do a verify signature of the digital signature on the QM1 certificate. The digital signature on the QM1 certificate was presumably signed by the SubCA1 private key. It is beyond the scope of this session to go into the details of how the verify signature process works, but at a high level the public key decrypts the digital signature back to a message digest and a comparison is done to see if this digital signature was signed by the private key that ties to the public key on the SubCA1 singer certificate. So it is critical here that you properly trust this SubCA1 signer certificate, as its public key is being used in the verify signature process.

Notes From Previous Slide - continued

- Assuming the verify signature of the QM1 certificate passed, the JavaClient now checks if the Issuer on the SubCA1 ceritificate matches the Subject. It does not (Issuer is RootCA1), so another step is done to look for a signer certificate in the JavaClient keystore with a Subject of RootCA1.

- The RootCA1 signer certificate is found in the keystore, and its public key is used to successfully validate the digital signature on the SubCA1 certificate. The JavaClient now checks if the RootCA1 signer certificate has a Subject that matches the Issuer. It does (they are both RootCA1), so this is a self-signed certificate or root CA signer certificate and the validation process stops. The JavaClient has now validated that it can trust the QM1 certificate.

TLS 1.2 Handshake - Authenticate The Client



* Indicates optional or situation-dependent messages that are not always sent.

NOTES (for our example TLS_RSA_WITH_AES_128_CBC_SHA256):

- In general, it is optional whether the client needs to be authenticated.
- If required, server will authenticate the client's certificate with the server's trust store.
- Client must also prove it has the private key for this client certificate by signing data in the CertificateVerify.

Notes From Previous Slide

- The JavaClient having to send a certificate for our TLS_RSA_WITH_AES_128_CBC_SHA256 example is optional. However, we will assume it is required (i.e. the SVRCONN channel on QM1 has a SSLCAUTH(REQUIRED)).
- QM1 will request for JavaClient to send a certificate with the CertificateRequest message (bold red in above diagram).
- JavaClient will send its certificate from its keystore to QM1 with the Certificate message (bold red in above diagram). QM1 would authenticate the JavaClient certificate similar to how JavaClient authenticated QM1's certificate (see previous slide for more details).
- JavaClient is also required to prove that it has the private key for this certificate by creating a digital signature with the private key and sending it in the CertificateVerify (bold red in above diagram). QM1 will then validate this digital signature with the public key that is included in the JavaClient certificate.
- We will see that for this TLS_RSA_WITH_AES_128_CBC_SHA256 example, QM1 does not have to do a CertificateVerify since QM1 will prove that it has the private key for QM1 public key through the process of sharing the pre-master secret. In other words, QM1 would not have been able to decrypt the pre-master secret that JavaClient encrypted with QM1's public key, if QM1 did not have the private key.

TLS 1.2 Handshake - Authenticate The Data

```
Client                                               Server

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                             <--------      ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                     -------->
                                          [ChangeCipherSpec]
                             <--------             Finished
Application Data             <------->     Application Data
```

* Indicates optional or situation-dependent messages that are not always sent.

NOTES (for our example TLS_RSA_WITH_AES_128_CBC_SHA256):

- A MAC (Message Authentication Code) is included with each encrypted message sent between client and server.
- The MAC is generated with the HMAC-SHA256 and the appropriate negotiated symmetric key.
- Each end can validate the MAC because both sides have built the same symmetric keys.
- “TLS is a cryptographic protocol which enables two parties (client and server) to identify and authenticate each other and communicate with confidentiality and data integrity.” - We have now accomplished this!

Notes From Previous Slide

- We have now covered most of the TLS 1.2 handshake. As a note, the ChangeCipherSpec messages note that future data between the TLS Client/Server is changing in how they will be encrypted, and the Finished messages are a way of validating that no outside entity has tampered with the TLS 1.2 handshake.

- We are now at the record protocol level. JavaClient and QM1 are now able to send encrypted data to each other. For our TLS_RSA_WITH_AES_128_CBC_SHA256 example, JavaClient and QM1 agreed on AES_128_CBC symmetric encryption to encrypt this data.

- However, there is also the need to put a MAC (Message Authentication Code) on each encrypted data record between the TLS Client/Server, to authenticate the data record and also be able to detect if the data has been tampered with.

- How the MAC is formed is beyond the scope of this session. At a high level and for this specific CipherSuite, the HMAC-SHA256 algorithm (which is what SHA256 in the last part of our CipherSuite name represents) is used to form this MAC. This HMAC-SHA256 algorithm also involves using one of the generated symmetric keys.

Notes From Previous Slide - continued

- Also, TLS 1.2 in general uses a MAC then encrypt approach to authenticate the record data and provide data integrity. This means that first a MAC is created, and then the plaintext data and MAC is encrypted to form the encrypted data record. We will see later in this session how TLS 1.3 moves away from the MAC then encrypt approach to AEAD (Authenticated Encryption with Associated Data) and why that was done.

- We stated earlier that “TLS is a cryptographic protocol which enables two parties (client and server) to identify and authenticate each other and communicate with confidentiality and data integrity.” was the purpose of TLS. We have now seen how the certificate validation process and sharing of the pre-master secret provide identification and authentication of the TLS Client and Server. The symmetric encryption and MAC provide the ability for the TLS Client and Server to communicate with confidentiality and data integrity. The MAC also authenticates the encrypted record data.

- The TLS_RSA_WITH_AES_128_CBC_SHA256 CipherSuite that we have been using for this example is called static RSA. It is static because you can see that the private/public key from the TLS Server is always used to protect the pre-master secret for each TLS session. However, this means that if anyone was recording the TLS session data, they could decrypt the encrypted data records if they are able to get ahold of the private key at some time in the future. For example, they could be capturing the session data for months, and then get ahold of the private key and decrypt all of that data that was previously captured. In an upcoming example, we will look at forward secrecy which prevents this type of security exposure.

CipherSuites

As we have seen now, the name of a TLS 1.2 CipherSuite defines some of the encryption algorithms that are used in the TLS 1.2 handshake.

Ex: CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256 specifies:

1) Key exchange and authentication algorithm used during handshake (asymmetric RSA)

2) Encryption algorithm used to encipher the data records (symmetric AES_128_CBC)

3) Algorithm for generating the MAC on the encrypted data records (HMAC-SHA256)

IBM MQ Note: This presentation exclusively uses the term CipherSuite (and not CipherSpec) as CipherSuite is the correct term to use from SSL v3 and going forward. CipherSpec in this context is an SSL v2 term that was replaced by CipherSuite at SSL v3 in the IETF TLS specifications.

CipherSuite and Digital Certificate Dependencies

- Not all IBM MQ CipherSuites may be used with all digital certificates.
- IBM MQ TLS 1.2 CipherSuites that start with ECDHE_ECDSA_ would use a certificate with an elliptic curve public key and digital signature.
- The other IBM MQ TLS 1.2 CipherSuites would use a certificate with an RSA public key and digital signature.
- To use the CipherSuite ECDHE_ECDSA_AES_128_CBC_SHA256, you would generate a certificate with elliptic curve keys and digital signature using runmqakm with options as follows:

```
runmqakm -certreq -create -sigalg SHA256WithECDSA -size 256 -db key.kdb -label
ibmwebspheremqQM2 -dn "CN=QM2" -file CertReqQM2.arm
```

- To use the CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256, you would generate a certificate with RSA keys and digital signature using runmqakm with options as follows:

```
runmqakm -certreq -create -sigalg SHA256WithRSA -size 2048 -db key.kdb -label
ibmwebspheremqQM1 -dn "CN=QM1" -file CertReqQM1.arm
```

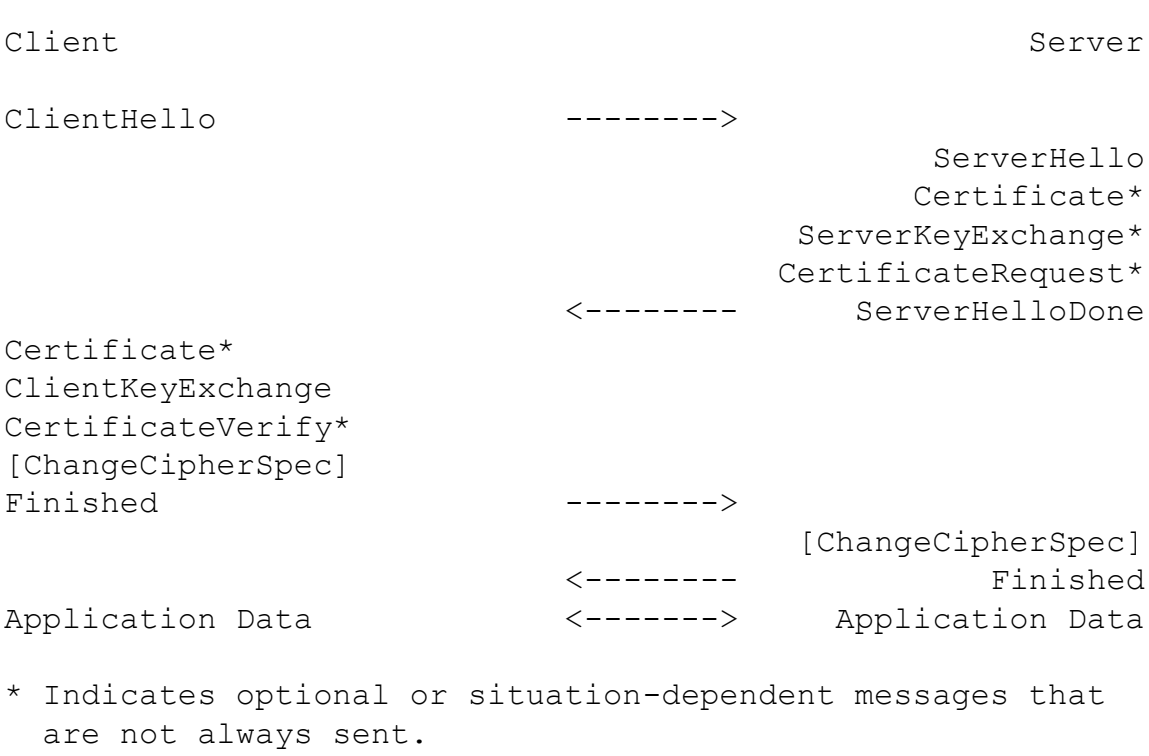
Certificate Request With RSA Key/Signature

```
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: CN=QM1
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:9a:76:bb:34:bd:4c:8f:8e:02:84:d2:8c:3d:79:
                    df:bf:30:c5:bd:99:bb:36:32:13:c7:36:c5:0b:f3:
                    .
                    fd:5a:1a:bb:86:55:a0:6c:e8:e8:44:d1:91:8b:ee:
                    c1:0b
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha256WithRSAEncryption
         0b:1c:8f:cb:d4:50:3d:9d:8b:09:bd:40:c9:ef:c7:3a:f2:54:
         a2:0f:f9:8b:38:24:10:8d:de:3a:1b:51:bc:d3:29:2a:29:e8:
         .
         f9:68:80:94:d1:ed:45:ac:09:6d:6d:5b:84:ae:7f:39:01:9f:
         f3:7f:91:a1
```

Certificate Request with Elliptic Curve Key/Signature

```
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: CN=QM2
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:79:45:1f:da:4c:d8:f8:46:7b:da:16:f4:df:8e:
                    e2:4a:05:50:eb:19:0a:c0:ac:ba:e1:76:28:4b:fe:
                    11:09:96:32:77:71:dd:ac:52:fe:ef:7f:57:2f:26:
                    11:d5:d5:fb:7a:8c:a8:62:87:51:af:28:8a:06:47:
                    24:0b:13:b2:6b
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        Attributes:
            a0:00
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:21:00:b5:df:f9:65:0d:eb:b1:b9:fa:da:e1:71:ad:
         4d:a9:04:24:11:a4:af:dd:8b:77:ba:ec:de:63:fe:8c:b3:e0:
         4c:02:20:3b:0f:b0:49:86:1e:c3:74:d1:81:83:e5:51:ef:a0:
         ce:f2:bc:19:d0:d5:b2:38:d4:7a:32:d3:3b:82:0f:00:fd
```

TLS 1.2 Handshake With ECDHE_RSA



* Indicates optional or situation-dependent messages that are not always sent.

Assume our chosen CipherSuite is ECDHE_RSA_AES_128_CBC_SHA256.

Notes about ECDHE_RSA:

1) Uses ECDHE to create temporary elliptic curve keys for the client/server at runtime to negotiate the symmetric keys.

2) Uses RSA certificates only for authentication.

3) ECDHE has forward secrecy. This means that even if someone gets your private key from your Server keystore, they can not decrypt your data that you have sent in your TLS session.

Notes From Previous Slide

- In this example we look at a TLS 1.2 handshake that supports forward secrecy. The CipherSuite here is ECDHE_RSA_AES_128_CBC_SHA256. It supports forward secrecy because each new TLS session uses newly generated public/private keys to share the pre-master secret between the TLS Client and Server. If anyone was able to get the TLS Server's private key, they would not be able to decrypt any of the TLS session data since this private key was not used to protect the pre-master secret share.
- The ECDHE piece of the CipherSuite stands for Elliptic Curve Diffie-Helman Ephemeral, where the ephemeral stands for temporary keys. ECDHE is used to protect the pre-master secret share between the TLS Client and Server and uses temporary public/private elliptic curve keys in this process.
- The RSA piece means that RSA public key certificates are used for authentication only.
- AES_128_CBC is the symmetric encryption that is used to protect the data records.
- HMAC_SHA256 is used to provide authentication and data integrity for the data records.
- For this ECDHE CipherSuite, the TLS Server has to send a digital signature signed by its private key in the ServerKeyExchange message to prove that it has the private key for the public key shared in the TLS Server certificate.

TLS 1.3

- TLS 1.3 (RFC 8446) was published in August 2018.
- TLS 1.3 supports five CipherSuites with a meaning of TLS_AEAD_HASH where the AEAD is used for record protection and the HASH to be used for HKDF (used for generating the symmetric keys):

  TLS_AES_128_GCM_SHA256

  TLS_AES_256_GCM_SHA384

  TLS_CHACHA20_POLY1305_SHA256

  TLS_AES_128_CCM_SHA256

  TLS_AES_128_CCM_8_SHA256

- AEAD stands for Authenticated Encryption with Associated Data. An example from above would be AES_128_GCM. Some TLS 1.2 CipherSuites (e.g. TLS_RSA_WITH_AES_128_CBC_SHA256) use a MAC-then-encrypt approach for record protection. AEAD uses an API approach for record protection.

Notes From Previous Slide

- AEAD is now required for CipherSuites at TLS 1.3.
- TLS 1.2 in general used a MAC-then-encrypt approach to provide confidentiality, authentication, and data integrity for the data record. The sending side would first create a MAC (e.g. using HMAC-SHA256). Then the plaintext data and MAC would be encrypted into an encrypted data record that was sent to the receiving side.
- This MAC-then-encrypt approach has led to security vulnerabilities within TLS, due to TLS implementers making mistakes in how they implemented MAC-then-encrypt. TLS is complicated to understand, and it is not always easy for TLS implementers to implement it correctly.
- AEAD uses an API approach where the TLS implementer can call an API to encrypt and decrypt the record. This API also provides the properties of confidentiality, authentication, and data integrity for the data record. If the data record has been tampered with, the decryption API call will fail with an error. Since an API approach is being used for AEAD, it is easier for a TLS implementer to implement it and avoid potential security vulnerabilities in the implementation process.

TLS 1.3

- Static RSA and Diffie-Hellman CipherSuites have been removed; all public-key based key exchange mechanisms now provide forward secrecy through a form of ephemeral Diffie-Hellman (e.g. ECDHE). As a consequence, the “backdoor” of static RSA is no longer available with TLS 1.3.

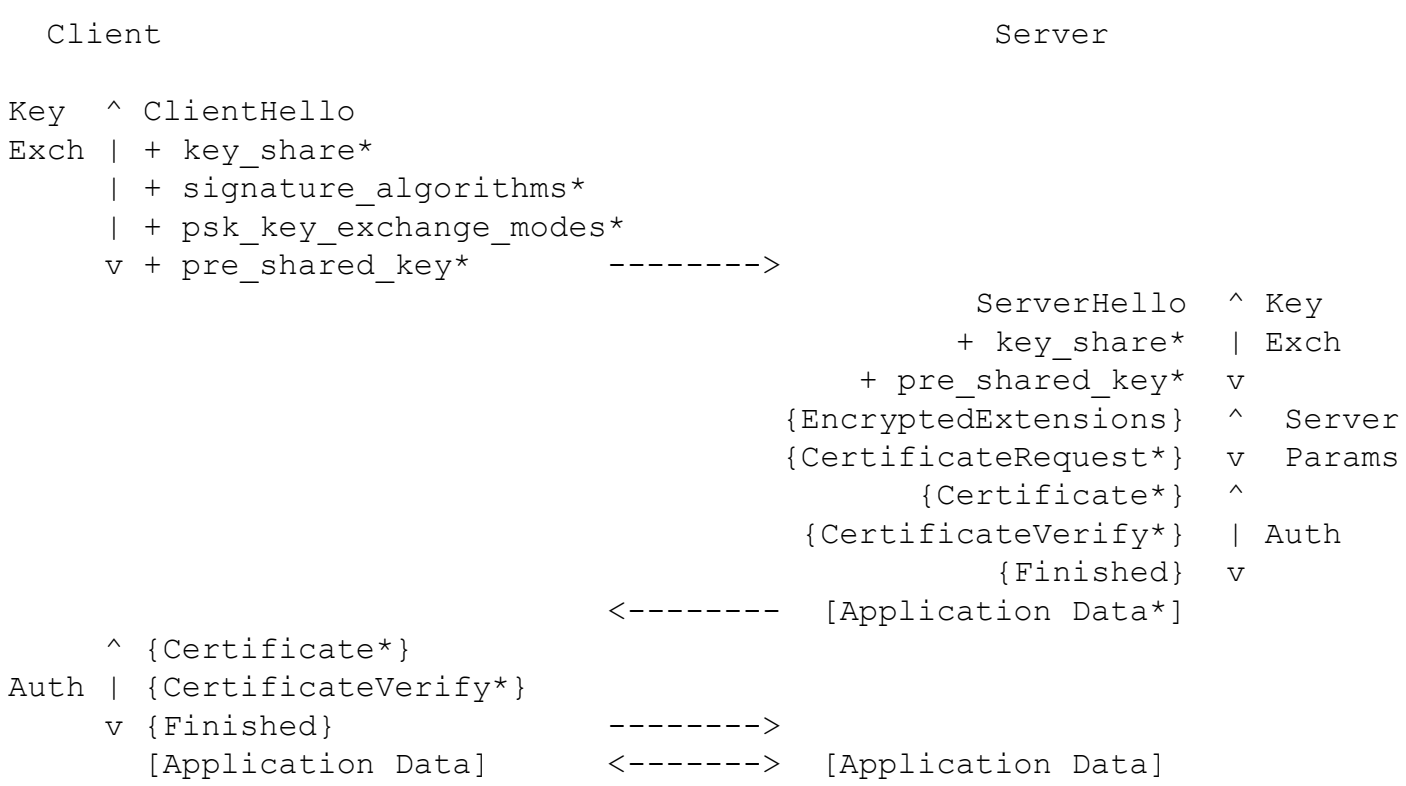
- RTT stands for Round Trip Time. TLS 1.3 reduces the handshake from 2-RTT (TLS 1.2) to 1-RTT (TLS 1.3) for new connections. For resuming connections, TLS 1.3 has a 0-RTT option (with a weaker security warning). This all results in TLS 1.3 having some potential performance improvements over TLS 1.2.

Notes From Previous Slide

- We discussed earlier in the presentation some of the security differences between static RSA and forward secrecy CipherSuites like ECDHE. For public-key based key exchange mechanisms, forward secrecy is a requirement at TLS 1.3.

- TLS 1.3 has some potential performance improvements over TLS 1.2 in the number of round trips. The number of round trips to start a new connection was reduced from 2 to 1 in TLS 1.3. There is also a zero round trip resumption option at TLS 1.3, but it does require the application to meet certain replay attack prevention requirements to be safe to use.

TLS 1.3 Handshake from RFC 8449



Notes From Previous Slide

- The TLS 1.3 handshake has changed fairly significantly between TLS 1.2 and TLS 1.3.
- Assuming PSK (Pre-Shared Keys) is not being used, a type of DHE (e.g. ECDHE) is used to share the pre-master secret between the TLS Client and Server.
- You can use certificates that have RSA public keys at TLS 1.3. However, static RSA can not be used to share the pre-master secret, since it does not provide forward secrecy.

openssl s_client

- The next few slides talk about a helpful TLS tool called openssl.
- openssl is a cryptography toolkit (command line tool on Unix) implementing the SSL and TLS network protocols and related cryptography standards required by them.
- s_client option for openssl implements a generic SSL/TLS client which can establish a transparent connection to a remote server speaking SSL/TLS. It's intended for testing purposes only and provides only rudimentary interface functionality but internally uses mostly all functionality of the OpenSSL ssl library.
- I have found it useful for the following when working with TLS and MQ:

a. Request that a TLS server (e.g. queue manager) send its personal certificate.

b. Validate what signer certificates a TLS server (e.g. queue manager) is currently using.

c. Retreive the signer certificates that a TLS server (e.g. LDAP server) is using.

openssl s_client Example

```
> openssl s_client -connect server1:1414 -showcerts –servername
tcz2e-chl1.chl.mq.ibm.com
```

- The above command will get the personal certificate and signer certificates for the queue manager running on server1 and listening on port 1414 and for channel TCZ.CHL1.
- Starting at MQ v8, a channel can have its own CERTLABL. Therefore, you do have to potentially be specific about which channel you are working with, if this queue manager is using multiple personal certificates.
- The oddly named tcz2e-ch1.chl.mq.ibm.com does tie to MQ channel TCZ.CHL1. IBM MQ documents how to determine the servername for a channel in the following support document.

  http://www-01.ibm.com/support/docview.wss?uid=swg21978780&myns=swgws&mynp=OCSSFKSJ&mync=R&cm_sp=swgws-_-OCSSFKSJ-_-R

openssl - Other Uses

- openssl can be used to create your own sandbox Certificate Authority. I used an “OpenSSL Cookbook” document from www.feistyduck.com to do this.

- openssl can display contents of PEM files.

```
> openssl x509 -in qm1.pem -noout -text
```

- openssl can create a private/public key pair and certificate request.

```
> openssl genrsa -aes128 -out fd.key 2048
> openssl req -new -key fd.key -out fd.csr
```

- And a myriad of other options as well for openssl! It is a great tool to have in your toolkit when working with TLS.

- This presentation is now complete. I hope you found it helpful!