TLS Trends

There has recently been heated discussion regarding TLS (Transport Layer Security) in the IETF (Internet Engineering Task Force). The reason the debate has become so lively is without a doubt the disclosure of the U.S. government's top-secret PRISM surveillance and intelligence gathering program by Edward Snowden in 2013. With the existence of the pervasive monitoring typified by PRISM being brought to light, the IETF was forced to revisit the issue of privacy. As stated in RFC 7258, protocols drafted by the IETF in the future will be designed to make pervasive monitoring more difficult.

As far as HTTP is concerned, the use of TLS will likely be highly recommended. In fact, when using HTTP/2 protocol that was published in 2015 (RFC 7540), the use of TLS is effectively mandatory because major browsers will require it. Of course, use of TLS is also strongly recommended with the current mainstream HTTP/1.1 protocol. HTTP servers must have certificates to use TLS. Until now the issuing of certificates cost money, and that discouraged many people from using TLS. It is now also possible to issue certificates for free due to the Let's Encrypt project.

The latest version of TLS is 1.2, and eight years have passed since it was standardized. Over this period of time, a variety of attack techniques that target TLS have been discovered. RFC 7457 is an outstanding document that provides a summary of these attack techniques. As new attack methods have appeared and various cryptographic technologies have become obsolete, the recommended methods for using TLS have also changed. The currently recommended methods are detailed in RFC 7525. The IETF is now working on the draft for TLS 1.3 based on this knowledge. In this article we will give an explanation of trends in TLS targeted at those who already know its mechanisms.

3.1 Versions

The previous incarnation of TLS was the SSL (Secure Socket Layer) protocol designed by Netscape Communications. The SSL 2.0 specification was published in 1995, and SSL 3.0 was published in 1996. There were a variety of issues with the design of SSL 2.0, and use of it was prohibited by RFC 6176. SSL 3.0 also had issues with attacks such as the POODLE vulnerability, as well as flaws in its design, and RFC 7568 required that it not be used.

SSL was brought in to the IETF and standardized, at which point it became TLS. TLS versions 1.0, 1.1, and 1.2 have been established. I will go into more detail later, but currently the use of a method called AEAD (Authenticated Encryption with Associated Data) is recommended for data authentication and encryption. AEAD cannot be used with TLS 1.0 or 1.1. Getting straight to the point, to use TLS safely it is now necessary to utilize TLS 1.2 via a suitable method.

Table 1 shows a summary of information regarding SSL/TLS versions (ID is an abbreviation of Internet-Draft). We will also discuss TLS 1.3 in this article, but as its specifications are currently being drawn up, please understand that it may end up slightly different.

Version	Specification	Year Established	Usage
SSL 2.0	Stopped at ID	1995	Use prohibited by RFC 6176
SSL 3.0	RFC 6101	1996 (RFC issued in 2011)	Use prohibited by RFC 7568
TLS 1.0	RFC 2246	1999	△
TLS 1.1	RFC 4346	2006	△
TLS 1.2	RFC 5246	2008	○
TLS 1.3	ID	Draft underway	

Table 1: SSL/TLS Versions

3.2 Suitable Cipher Suites

RFC 5246 in which TLS 1.2 was established requires that the TLS_RSA_WITH_AES_128_CBC_SHA cipher suite be implemented. This has the following meaning:

- RSA for key exchange
- RSA also for server authentication
- AES in CBC mode for the encryption of communications
- SHA1 for the MAC generation function

The TLS 1.2 cipher suite required by HTTP/2 and recommended for first proposal in RFC 7525 is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256. This is interpreted as follows:

- ECDHE (Elliptic Curve Diffie-Hellman, Ephemeral) for key exchange
- RSA for server authentication
- AES 128 in GCM (Galois/Counter Model) mode for the encryption of communications
- SHA256 for the secure hash function

In TLS 1.3, the implementation of TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 is required in addition to TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256. Server authentication has merely changed from RSA to ECDSA (DSA using elliptic curve cryptography).

Next, I will explain the reasons why the recommended cipher suites have changed in this way.

3.3 Public-Key Cryptography and Key Exchange

As mentioned above, the recommended key exchange method has changed from RSA to ephemeral Diffie-Hellman keys. This is because ephemeral Diffie-Hellman keys provide forward secrecy, which RSA lacks. Forward secrecy means that the secrecy of data is protected going forward.

Let us look at why there is no forward secrecy when RSA is used for key exchange. When a client connects to a server using TLS, the server sends a certificate to the client. This certificate consists of the server's public key signed by a certificate authority. The client generates the secret it needs to share with the server, and encrypts it with the server's RSA public key before sending it to the server. Only a server with the RSA private key can decrypt this ciphertext. This means the client and server have successfully shared secret data, so symmetric-key cryptography is used to protect the communication channel using a key generated from this secret.

At this point, the encrypted channel is secure. It is almost impossible for a third party to intercept content. However, if the following events were to actually occur, communications could be intercepted.

Let us suppose that certain pervasive monitoring has captured all the data carried over this encrypted channel. Then, when the server is destroyed due to it being swapped out, the data on the hard disk was not erased by mistake. If the party performing pervasive monitoring were to obtain this hard disk, it could extract the private key, so it would be possible to decrypt the saved encrypted channel data in sequence.

Meanwhile, with ephemeral Diffie-Hellman methods, the client and server both generate temporary public and private keys. These private keys are not saved to the hard disk, so the aforementioned example would not happen.

It seems that the ECDHE (RFC 4492) ephemeral Diffie-Hellman method achieved through elliptic curve cryptography is more likely to see widespread use than the original DHE (Diffie-Hellman, Ephemeral). This is because of the following:

- ECDHE exchanges less data than DHE.
- The computation rate of ECDHE is faster than DHE.
- ECDHE has carefully-selected parameters defined in advance. Although there is an ID aimed at defining parameters for DHE, it has not yet reached the RFC stage.
- As mentioned earlier, ECDHE is listed as the suite for first proposal in RFC 7525.

See "1.4.2 Forward Secrecy" in IIR Vol.22 for more information about forward secrecy.

3.4 The Obsolescence of Symmetric-Key Cryptography

TLS 1.1 and earlier uses the following two ciphertext formats.

- Stream ciphers
- CBC (Cipher Block Chaining) mode block ciphers

A range of attack methods have been found in RC4, which is the only practical option for stream ciphers, so their use is prohibited (RFC 7465).

With regard to TLS 1.0 and earlier CBC mode block ciphers, the BEAST attack method is well known. Additionally, the "MAC-then-encrypt" method is used with TLS 1.2 and earlier CBC mode block ciphers. A MAC (Message Authentication Code) is auxiliary data for ensuring data has not been altered and authenticating it. MAC-then-encrypt involves generating a MAC from plaintext, then appending this MAC to the plaintext and encrypting the result. An attack technique called padding oracle attacks that targets MAC-then-encrypt has been found. For this reason, the "encrypt-then-MAC" format is proposed in RFC 7366 as a replacement for MAC-then-encrypt.

In TLS 1.2, AEAD (Authenticated Encryption with Associated Data) was specified as a third format for ciphertext. AEAD is a method in which encryption and authentication are carried out simultaneously. Currently, the use of AEAD is recommended instead of stream ciphers or CBC mode block ciphers. The following symmetric-key encryption modes can be used with AEAD:

- AES-GCM (Galois/Counter Model) mode
- AES-CCM (Counter with CBC-MAC) mode

In TLS 1.3, the stream cipher and CBC mode block cipher formats have been deleted, so only AEAD is defined.

3.5 Handshake

In this section, I will explain actual TLS communications.

3.5.1 Full Handshake

When a client first connects to a server, a full handshake must be performed. In TLS 1.2, the process shown in Figure 1 takes place when TLS_RSA_WITH_AES_128_CBC_SHA is selected.

- The client advertises the cipher suites it supports in a ClientHello message.
- The server indicates it has selected TLS_RSA_WITH_AES_128_CBC_SHA in a ServerHello message. The server's RSA certificate is included in the Certificate message.
- The client generates a secret, encrypts it with the server's RSA public key, and sends it stored in a ClientKeyExchange message. A ChangeCipherSpec message is then sent to switch the communication channel to an encrypted channel. This channel is encrypted using AES-CBC mode. Immediately after switching to an encrypted channel, a Finished message is sent to confirm that the handshake concluded successfully. All data subsequently received from applications is also sent using this encrypted channel. The gray parts of Figure 1 indicate the encrypted channel.

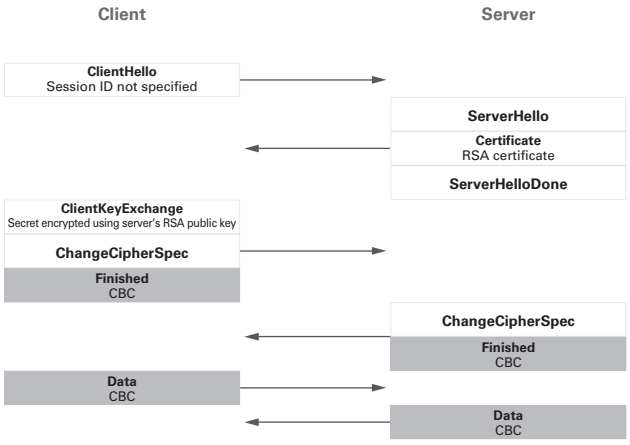

Figure 1: TLS 1.2 Full Handshake TLS_RSA_WITH_AES_128_CBC_SHA

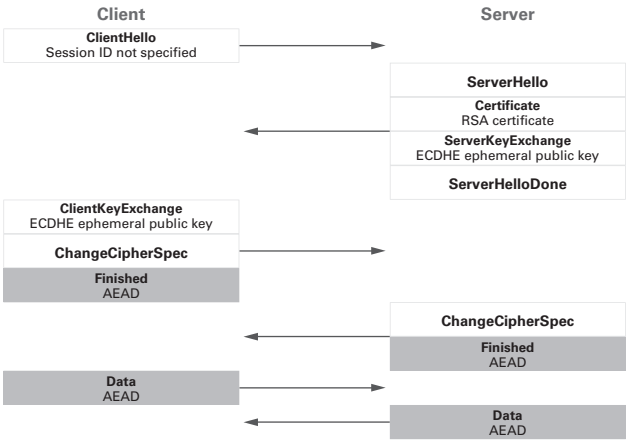

Figure 2: TLS 1.2 Full Handshake TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

- The server uses its private key to extract the secret, and in the same way as the client sends a ChangeCipherSpec message to switch the communication channel to an encrypted channel.

Next, I will discuss what happens when TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 is selected in TLS 1.2 (Figure 2). The differences compared to TLS_RSA_WITH_AES_128_CBC_SHA are as follows:

- After the server receives the ClientHello, it selects TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256. Then, it generates ECDHE ephemeral public and private keys. The public key is sent inside a ServerKeyExchange message.
- The client also generates ECDHE ephemeral public and private keys. It generates a secret from its private key and the server's public key. Its public key is sent inside a ClientKeyExchange message.
- The server generates a secret from its private key and the client's public key.

The full handshake process doesn't change at all for TLS 1.0, 1.1, or 1.2. However, the full handshake process has been radically redesigned in TLS 1.3. Above all, it reduces the RTT (Round Trip Time) by one step by handling key exchange in the Hello message.

- The client creates ECDHE ephemeral public and private keys, and sends the public key stored in a ClientHello message option.
- The server also creates ECDHE ephemeral public and private keys, and sends them stored in a ServerHello message option. The communication channel is immediately encrypted from this point. The Certificate and Finished messages that store server certificates are sent encrypted. After the Finished message is sent, a switch is made to an even more secure encrypted channel. The different shades of gray in Figure 3 indicate these different encrypted channels.
- After sending a Finished message over the current encrypted channel, the client switches to a more secure encrypted channel.

3.5.2 Resuming a Session

Once a client and server have performed a full TLS 1.2 handshake, the key exchange process can be omitted by resuming that session. Take a look at Figure 4.

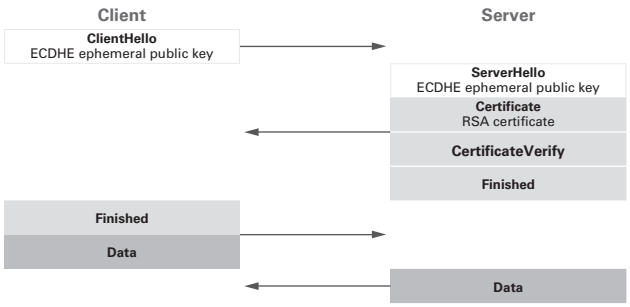

Figure 3: TLS 1.3 Full Handshake TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

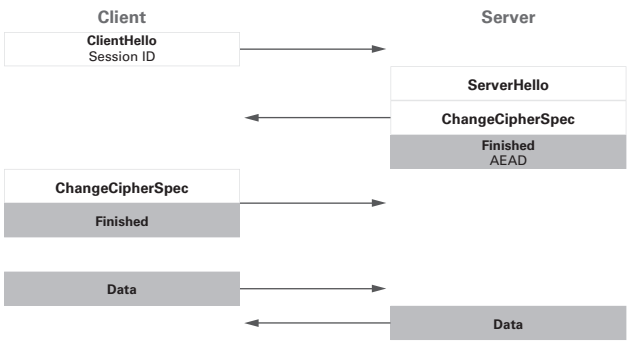

Figure 4: TLS 1.2 Session Resumption

- The client specifies the session ID for the session it would like to resume in the ClientHello message.
- If the server has saved the status of the specified session ID, it uses this to switch to an encrypted channel.

In addition to eliminating the need for heavy public key cryptography calculations, resuming a session also reduces the RTT by one step. However, this method requires that

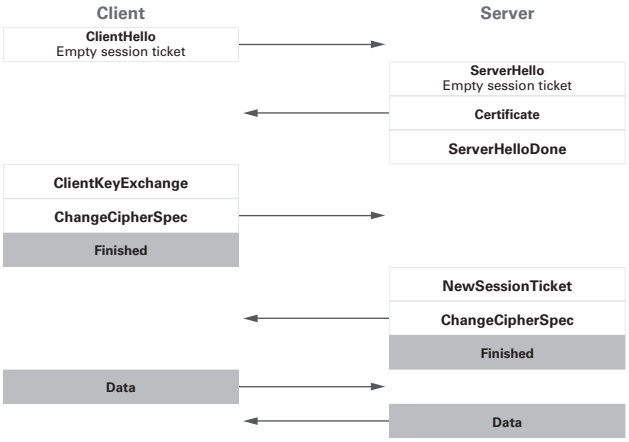

Figure 5: Full Handshake for TLS 1.2 Session Tickets

servers save session information. The amount of status data that must be retained also increases in proportion to the number of clients. This method increases the load on the server, which isn't exactly ideal.

The session ticket method defined in RFC 5077 is one way to reduce this server load. Session tickets are encrypted session information that only the server can decrypt. Using session tickets does away with the need for servers to retain session information. To use session tickets with TLS 1.2, it is first necessary to perform a full handshake for the session tickets (Figure 5).

- The client sends an empty session ticket as an extension of the ClientHello message to notify the server that it supports session tickets.
- The server also notifies the client that it supports session tickets by specifying an empty session ticket in the ServerHello options.
- Immediately before switching to an encrypted channel using ChangeCipherSpec, the server sends the session ticket it has generated in a NewSessionTicket message.
- The client associates the session ticket it was sent with the current session information, then saves it.

Next, I will explain how to resume a session using session tickets in TLS 1.2 (Figure 6).

- The client extracts the session information and session ticket to resume, and sends the session ticket in the ClientHello options.
- The server decrypts the session ticket to obtain the session information. If necessary, new session information is sent in a NewSessionTicket message. The server then switches to an encrypted channel.
- The client uses the aforementioned session information to switch to the encrypted channel.

TLS 1.3 session tickets are integrated with the PSK (Pre-Shared Key) defined in RFC 4297. The PSK method involves the use of a pre-shared secret instead of a public key for server or client authentication. If the TLS 1.3 PSK handshake is only used for the session ticket function, it doesn't differ much from TLS 1.2 (Figure 7). The minor differences are as follows:

- After a full handshake, the server can send NewSessionTicket messages at any time.
- As with the TLS 1.3 full handshake, the encrypted channel switches twice.

3.5.3 Client Authentication Using Certificates

Let us consider a case in which a certain server is accessing a certain page using TLS. We will assume the links on that page all point to the same server, but the content requires certificate-based client authentication.

In TLS 1.2, renegotiation is carried out when certificate-based client authentication becomes necessary at some point. This involves performing the handshake process again. Unlike a full handshake, this handshake is performed within the encrypted channel (Figure 8).

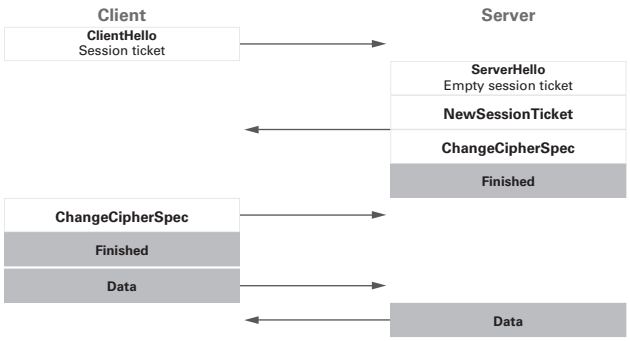

Figure 6: Resuming a Session Using TLS 1.2 Session Tickets

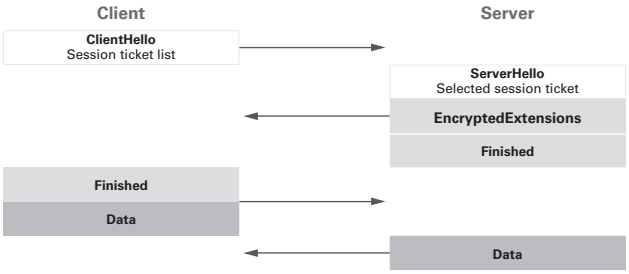

Figure 7: Resuming a Session Using TLS 1.3 Session Tickets

- The server sends a HelloRequest message to the client requesting renegotiation.
- The client sends a ClientHello message.
- The server sends a CertificateRequest along with the ServerHello message requesting the client's certificate.
- The client sends the client certificate in a Certificate message when sending the ClientKeyExchange message.

The original purpose of renegotiation is to refresh and extend the life of the encrypted channel. It is also used for certificate-based client authentication due to the limitation in TLS 1.2 that requires CertificateRequest messages to be sent immediately after the ServerHello message.

In TLS 1.3, a clear distinction is made between refreshing the encrypted channel and certificate-based client authentication. This enables CertificateRequest messages to be sent from the server to the client at any time (Figure 9).

3.5.4 0-RTT

In TLS 1.3, a handshake mode called 0-RTT that also encrypts and sends application data when sending a ClientHello message is under consideration. This is a little complicated, so I'll skip the explanation for now. Anyone interested should refer to the TLS 1.3 ID.

3.6 Compression

In TLS 1.2 and earlier, there is a compressed text format in addition to plaintext and ciphertext. When using a compression function, plaintext is compressed into compressed text, then this is encrypted to create ciphertext. Unfortunately, when a compression function is used, the text is vulnerable to attacks such as CRIME and BREACH.
For this reason, you cannot use compression functions when using TLS 1.2. Encrypt plaintext directly to create the ciphertext. In TLS 1.3, the compressed text format has been deleted, and only plaintext and ciphertext are defined.

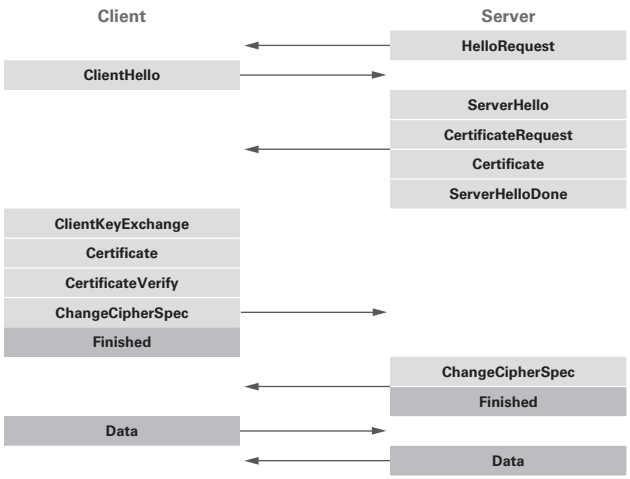

Figure 8: Certificate-Based Client Authentication Using TLS 1.2

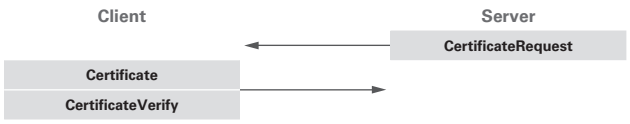

Figure 9: Certificate-Based Client Authentication Using TLS 1.3

3.7 Let's Encrypt

Let's Encrypt is a project for automatically issuing free server certificates. Only Domain Validation (DV) certificates can be issued, so the issuing of Organization Validation (OV) or Extended Validation (EV) certificates is not possible. At this point in time, wildcard certificates cannot be issued. When there are multiple server names, you can either request the issue of enough DV certificates to match the number of servers, or you can use a Subject Alternative Name (SAN). The commands provided by Let's Encrypt implement the ACME (Automatic Certificate Management Environment) protocol that the IETF is currently working on standardizing. See "1.4.2 The Let's Encrypt Project and the ACME Protocol for Automatic Certificate Issuing" in IIR Vol.30 for more information about Let's Encrypt.

3.8 Final Remark

In this article, I have only given the names of attack techniques, and not explained the specific methods involved. More detailed explanations of each attack technique can be found easily by searching online. Anyone interested should look up this information to find out more.

Author:
Kazuhiko Yamamoto
Senior Researcher, Research Laboratory, IIJ Innovation Institute Inc.
Mr. Yamamoto is interested in applying the parallel technology of the Haskell programming language to network programming.
Recently he has been working on the HTTP/2 and TLS 1.3 protocols.
He has translated the books "Programming in Haskell" and "Parallel and Concurrent Programming in Haskell".