Transport Layer Securities (TLS) are designed to provide security at the transport layer. TLS was derived from a security protocol called Secure Socket Layer (SSL). TLS ensures that no third party may eavesdrop or tampers with any message.

There are several benefits of TLS:

- **Encryption:**

  TLS/SSL can help to secure transmitted data using encryption.
- **Interoperability:**

  TLS/SSL works with most web browsers, including Microsoft Internet Explorer and on most operating systems and web servers.
- **Algorithm flexibility:**

  TLS/SSL provides operations for authentication mechanism, encryption algorithms and hashing algorithm that are used during the secure session.
- **Ease of Deployment:**

  Many applications TLS/SSL temporarily on a windows server 2003 operating systems.
- **Ease of Use:**

  Because we implement TLS/SSL beneath the application layer,

most of its operations are completely invisible to client.

Working of TLS:
The client connect to server (using TCP), the client will be something. The client sends number of specification:

1. Version of SSL/TLS.
2. which cipher suites, compression method it wants to use.

The server checks what the highest SSL/TLS version is that is supported by them both, picks a cipher suite from one of the clients option (if it supports one) and optionally picks a compression method. After this the basic setup is done, the server provides its certificate. This certificate must be trusted either by the client itself or a party that the client trusts. Having verified the certificate and being certain this server really is who he claims to be (and not a man in the middle), a key is exchanged. This can be a public key, "PreMasterSecret" or simply nothing depending upon cipher suite.

Both the server and client can now compute the key for symmetric encryption. The handshake is finished and the two hosts can communicate securely. To close a connection by finishing. TCP connection both sides will know the connection was improperly terminated. The connection cannot be compromised by this through, merely interrupted.

Transport Layer Security (TLS) continues to play a critical role in securing data transmission over networks, especially on the internet. Let's delve deeper into its workings and significance:

Enhanced Security Features:

TLS employs a variety of cryptographic algorithms to provide a secure communication channel. This includes symmetric encryption algorithms like AES (Advanced Encryption Standard) and asymmetric algorithms like RSA and Diffie-Hellman key exchange. Additionally, TLS supports various hash functions for message integrity, such as SHA-256, ensuring that data remains confidential and unaltered during transit.

Certificate-Based Authentication:

One of the key components of TLS is its certificate-based authentication mechanism. When a client connects to a server, the server presents its digital certificate, which includes its public key and other identifying information. The client verifies the authenticity of the certificate using trusted root certificates stored locally or provided by a trusted authority, thereby establishing the server's identity.

Forward Secrecy:

TLS supports forward secrecy, a crucial security feature that ensures that even if an attacker compromises the server's private key in the future, they cannot decrypt past communications. This is achieved by generating ephemeral session keys for each session, which are not stored and thus cannot be compromised retroactively.

TLS Handshake Protocol:

The TLS handshake protocol is a crucial phase in establishing a secure connection between the client and the server. It involves multiple steps,

including negotiating the TLS version, cipher suite, and exchanging cryptographic parameters. The handshake concludes with the exchange of key material used to derive session keys for encrypting and decrypting data.

Perfect Forward Secrecy (PFS):

Perfect Forward Secrecy is an advanced feature supported by TLS that ensures the confidentiality of past sessions even if the long-term secret keys are compromised. With PFS, each session key is derived independently, providing an additional layer of security against potential key compromise.

TLS Deployment Best Practices:

To ensure the effectiveness of TLS, it's essential to follow best practices in its deployment. This includes regularly updating TLS configurations to support the latest cryptographic standards and protocols, disabling deprecated algorithms and cipher suites, and keeping certificates up-to-date with strong key lengths.

Continual Evolution:

TLS standards continue to evolve to address emerging security threats and vulnerabilities. Ongoing efforts by standards bodies, such as the Internet Engineering Task Force (IETF), ensure that TLS remains robust and resilient against evolving attack vectors.

Conclusion:

In an increasingly interconnected world where data privacy and security are paramount, Transport Layer Security (TLS) serves as a foundational technology for securing communication over networks. By providing encryption, authentication, and integrity protection, TLS enables secure

data transmission, safeguarding sensitive information from unauthorized access and tampering. As cyber threats evolve, TLS will continue to evolve, adapting to new challenges and reinforcing the security posture of digital communications.

How does TLS work?

TLS uses a combination of symmetric and asymmetric cryptography, as this provides a good compromise between performance and security when transmitting data securely.

With symmetric cryptography, data is encrypted and decrypted with a secret key known to both sender and recipient; typically 128 but preferably 256 bits in length (anything less than 80 bits is now considered insecure). Symmetric cryptography is efficient in terms of computation, but having a common secret key means it needs to be shared in a secure manner.

Asymmetric cryptography uses key pairs – a public key, and a private key. The public key is mathematically related to the private key, but given sufficient key length, it is computationally impractical to derive the private key from the public key. This allows the public key of the recipient to be used by the sender to encrypt the data they wish to send to them, but that data can only be decrypted with the private key of the recipient.

The advantage of asymmetric cryptography is that the process of sharing encryption keys does not have to be secure, but the mathematical relationship between public and private keys means that much larger key sizes are required. The recommended minimum key length is 1024 bits, with 2048 bits preferred, but this is up to a thousand times more computationally intensive than symmetric keys of equivalent strength (e.g. a 2048-bit asymmetric key is approximately equivalent to a 112-bit symmetric key) and makes asymmetric encryption too slow for many purposes.

For this reason, TLS uses asymmetric cryptography for securely generating and exchanging a session key. The session key is then used for encrypting the data transmitted by one party, and for decrypting the data received at the other end. Once the session is over, the session key is discarded.

A variety of different key generation and exchange methods can be used, including RSA, Diffie-Hellman (DH), Ephemeral Diffie-Hellman (DHE), Elliptic Curve Diffie-Hellman (ECDH) and Ephemeral Elliptic Curve Diffie-Hellman (ECDHE). DHE and ECDHE also offer forward secrecy whereby a session key will not be compromised if one of the private keys is obtained in future, although weak random number generation and/or usage of a limited range of prime numbers has been postulated to allow the cracking of even 1024-bit DH keys given state-level computing resources. However, these may be considered implementation rather than protocol issues, and there are tools available to test for weaker cipher suites.

With TLS it is also desirable that a client connecting to a server is able to validate ownership of the server's public key. This is normally undertaken using an X.509 digital certificate issued by a trusted third party known as a Certificate Authority (CA) which asserts the authenticity of the public key. In some cases, a server may use a self-signed certificate which needs to be explicitly trusted by the client (browsers should display a warning when an untrusted certificate is encountered), but this may be acceptable in private networks and/or where secure certificate distribution is possible. It is highly recommended though, to use certificates issued by publicly trusted CAs.

What is a CA?
A Certificate Authority (CA) is an entity that issues digital certificates conforming to the ITU-T's X.509 standard for Public Key Infrastructures (PKIs). Digital certificates certify the public key of the owner of the certificate (known as the subject), and that the owner controls the domain being secured by the certificate. A CA therefore acts as a trusted third party

that gives clients (known as relying parties) assurance they are connecting to a server operated by a validated entity.

End entity certificates are themselves validated through a chain-of-trust originating from a root certificate, otherwise known as the trust anchor. With asymmetric cryptography it is possible to use the private key of the root certificate to sign other certificates, which can then be validated using the public key of the root certificate and therefore inherit the trust of the issuing CA. In practice, end entity certificates are usually signed by one or more intermediate certificates (sometimes known as subordinate or sub-CAs) as this protects the root certificate in the event that an end entity certificate is incorrectly issued or compromised.

Root certificate trust is normally established through physical distribution of the root certificates in operating systems or browsers. The main certification programs are run by Microsoft (Windows & Windows Phone), Apple (OSX & iOS) and Mozilla (Firefox & Linux) and require CAs to conform to stringent technical requirements and complete a WebTrust, ETSI EN 319 411-3 (formerly TS 102 042) or ISO 21188:2006 audit in order to be included in their distributions. WebTrust is a programme developed by the American Institute of Certified Public Accountants and the Canadian Institute of Chartered Accountants, ETSI is the European Telecommunications Standards Institute, whilst ISO is the International Standards Organisation.

Root certificates distributed with major operating systems and browsers are said to be publicly or globally trusted and the technical and audit requirements essentially means the issuing CAs are multinational corporations or governments. There are currently around fifty publicly trusted CAs, although most/all have more than one root certificate, and most are also members of the CA/Browser Forum which develops industry guidelines for issuing and managing certificates.

TLS: The technical details

TLS consists of many different elements. The fundamental part is the record protocol, the underlying protocol responsible for the overarching structure of everything else.

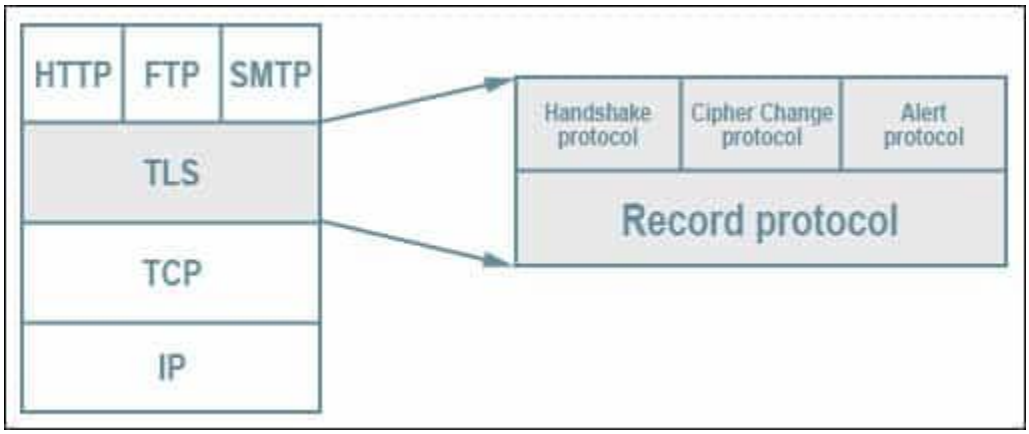

Diagram showing the TLS stack. TLS protocol stack by Jeffreytedjosukmono. Licensed under CC0.

The record protocol contains five separate subprotocols, each of which are formatted as **records**:

- **Handshake** – This protocol is used to set up the parameters for a secure connection.
- **Application** – The application protocol begins after the handshake process, and it is where data is securely transmitted between the two parties.
- **Alert** – The alert protocol is used by either party in a connection to notify the other if there are any errors, stability issues or a potential compromise.
- **Change Cipher Spec** – This protocol is used by either the client or the server to modify the encryption parameters. It's pretty straightforward, so we won't cover it in depth in this article.
- **Heartbeat** – This is a TLS extension that lets one side of the connection know whether its peer is still alive, and prevents firewalls

from closing inactive connections. It's not a core part of TLS, so we'll be skipping it in this guide.

Each of these subprotocols are used in different stages to communicate different information. The most important ones to understand are the handshake and the application protocols, because these are responsible for establishing the connection and then securely transmitting the data.

The TLS 1.2 handshake protocol

This is where the connection is established in a secure manner. It may seem complex if you are new to some of the concepts, but each of these are covered later on in the article if you need to refer to them.

There are three basic types of TLS handshake: the **basic TLS handshake**, the **client-authenticated TLS handshake** and the **abbreviated handshake**.

The basic TLS 1.2 handshake

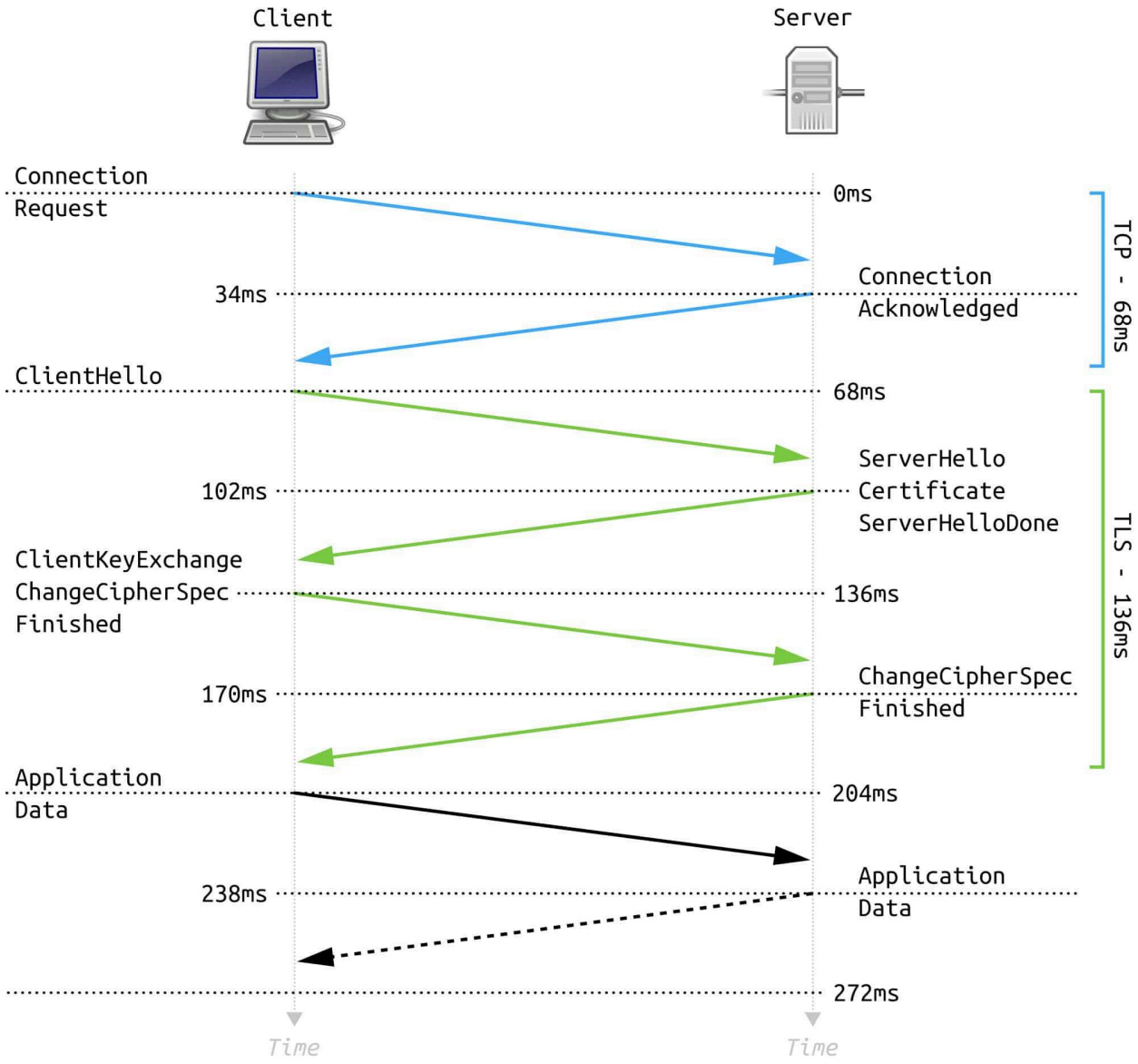

Diagram showing the TLS handshake process. Full TLS 1.2 Handshake by FleshGrinder. Licensed under CC0.

In this type of handshake, only the server is authenticated and not the client. It begins with the negotiation phase, where a client sends a **Client Hello** message. This contains the highest version of TLS that the client supports, possible cipher suites, an indication of whether it supports compression, a random number and some other information

The Client Hello message is met with a **Server Hello** message. This response contains the session ID, protocol version, cipher suite and compression (if any is being used) that the server selected from the client's list. It also includes a different random number.

It depends on the cipher suite that has been selected, but the server will generally follow this by sending a **Certificate** message for authentication. This validates its identity and contains its public key.

If ephemeral Diffie-Hellman or anonymous Diffie-Hellman key exchanges are being used, then this is followed by a **Server Key Exchange** message. Other key exchange methods skip this part. When the server has finished with its side of the negotiation, it sends a **Server Hello Done** message.

Now it's the client's turn again. Depending on the chosen cipher suite, it will send a **Client Key Exchange** message. This can contain a public key or a premaster secret, which is encrypted with the server's public key.

Both parties then use the random numbers and the premaster secret to come up with a master secret. Keys are derived from the master secret, which are then used to authenticate and encrypt the communications.

The client then sends a **Change Cipher Spec** message. This tells the server that the following messages will now be authenticated and encrypted (although sometimes encryption may not be used).

The client then follows this up with a **Finished** message, which is encrypted and also contains a Message Authentication Code (MAC) for authentication. The server decrypts this message and verifies the MAC. If any of these processes fail, then the connection should be rejected.

Now it's the server's turn to send a **Change Cipher Spec** message, as well as a **Finished** message with the same content as above. The client then also tries to decrypt and verify the contents. If this is all completed successfully, the handshake is finished. At this point, the application protocol is established. Data can then be exchanged securely in the same

way as the **Finished** message from above, with authentication and optional encryption.

Client-authenticated TLS handshake

This handshake is much like the basic TLS handshake, but the client is authenticated as well. The main difference is that after the server sends its **Certificate** message, it also sends a **Certificate Request** message, asking for the client's certificate. Once the server is finished, the client sends its certificate in a **Certificate** message.

The client then sends its **Client Key Exchange** message, just like in the basic TLS handshake. This is followed by the **Certificate Verify** message, which includes the client's digital signature. Since it is calculated from the client's private key, the server can verify the signature using the public key that was sent as part of the client's digital certificate. The rest of the **Client-authenticated TLS handshake** follows along the same lines as the basic TLS handshake.

Abbreviated TLS handshake

Once a handshake has already taken place, TLS allows much of the process to be cut out by using an abbreviated handshake instead. These handshakes use the session ID to link the new connection to the previous parameters.

An abbreviated handshake allows both parties to resume the secure connection with the same setup that was negotiated earlier. Because some of the cryptography that is normally involved in initiating a handshake can be pretty heavy on computational resources, this saves time and makes the connection easier.

The process begins with the **Client Hello** message. It's much like the earlier Client Hello message, but it also contains the **session ID** from the earlier connection. If the server knows the session ID, it includes it in its **Server Hello** message. If it does not recognize the session ID, it will return

a different number, and a full TLS handshake will have to take place instead.

If the server does recognize the session ID, then the **Certificate** and **Key Exchange** steps can be skipped. The **Change Cipher Spec** and **Finished** messages get sent in the same way as the basic TLS handshake shown above. Once the client has decrypted the message and verified the MAC, data can be sent across the secure TLS connection.

There is also a TLS extension that allows connections to be resumed with session tickets instead of session IDs. The server encrypts data about the session and sends it to the client. When the client wants to resume this connection, it sends the session ticket to the server, which decrypts it to reveal the parameters.

Session tickets aren't used as frequently, because they require the extension. Despite this, they can be advantageous in certain situations, because the server doesn't have to store anything.

Unpacking the TLS 1.2 handshake

The three most important steps of the handshake include:

- the parameters are selected,
- it conducts authentication, and
- the keys are established.

Let's cover them in a bit more detail so that you can understand what's really going on.

The parameters

At the start of the handshake, the client and the server negotiate the parameters of the connection by mutual agreement. The first of these is which version of TLS will be used. This is the highest version that both parties support, which tends to be the most secure.

The parties also decide which key exchange algorithm they will use to establish the master key. The hash function, encryption algorithm, and compression method are also agreed upon in this stage. These will be covered in detail when we discuss the **Application protocol** later in the article.

Authentication: Digital certificates

Authentication is a key part of securing a communication channel, because it lets both parties know that they are actually talking to who they think they are and not an impostor. In TLS and many other security mechanisms, this is achieved with what are known as digital certificates.

Digital certificates are electronic documents that show the link between an individual or entity and their public key. This link is validated by a certificate authority (CA), which is a trusted organization that verifies that the two are actually related, then uses its own reputation to grant trust to the certificate.

Different certificate levels represent varying degrees of trust. The important thing to know is that if a client or a server has a reliable and valid certificate, then it is reasonable to assume that the public key is legitimate and that you are not dealing with an attacker.

A note on public keys

Public-key encryption (also known as asymmetric encryption) is an important part of cryptography, and it is used extensively in the different aspects of TLS. Here's a quick primer for those who are unfamiliar with how it works.

The short explanation is that **public-key cryptography uses a pair of keys for encryption and decryption, rather than just a single key**.

The sender uses their intended recipient's public key to encrypt data. Once it has been encrypted, it can only be decrypted by the recipient's matching

private key. Of course, the public key can be shared publicly while the private key must be kept secret.

Public-key encryption allows parties to share information securely, even if they have never met or had an opportunity to exchange keys beforehand. It does this through some unique properties of prime numbers. Public-key encryption circumvents the problem inherent with symmetric cryptography; namely, how to securely exchange a single key.

Establishing a master secret

As we saw above when we discussed the process of the basic TLS handshake, after a party (or both parties) proves its identity with its public certificate, **the next step is to establish the master secret, also known as the shared secret**. The master secret is the base for deriving the keys used to both encrypt and check the integrity of data transmitted between the two parties.

The TLS handshake can use a number of different mechanisms to share this secret securely. These include RSA, several different types of the Diffie-Hellman key exchange, PSK, Kerberos, and others. Each has its own advantages and disadvantages, such as providing forward secrecy, but these differences are out of the scope of this article.

The exact process will depend on which key exchange method has been chosen, but it follows the rough steps mentioned in **The basic TLS handshake** section.

The premaster secret is derived according to whichever key exchange method had previously been selected. The client encrypts the premaster secret with the server's public key to safely send it across the connection.

The client and server then both use the premaster secret and the random numbers that they sent at the start of the communication to come up with the master secret. Once the master key has been calculated, it is used to come up with either four or six separate keys. These are the:

- **Client write MAC key** – This key is used by the server to authenticate data that is sent by the client.
- **Server write MAC key** – The server write MAC key is used by the client to authenticate the data that is sent by the server.
- **Client write encryption key** – This key encrypts data that the client writes.
- **Server write encryption key** – The server write encryption key encrypts the data that the server writes.
- **Client write IV key** – The client write IV key is generated when AEAD is used for encryption and authentication.
- **Server write IV key** – Similarly, the server write IV key is generated when AEAD is used for encryption and authentication.

Establishing the master key is an important part of the TLS handshake, because it enables both sides of the connection to securely derive keys that can be used for both authentication and encryption. Separate keys are used for both processes as a precaution.

Once the authentication and encryption keys have been derived, they are used to protect both **Finished** messages, as well as records sent through the application protocol.

The application protocol

Once a secure connection has been established by the TLS handshake, the application protocol is used to protect the transmitted data. It can be configured to use a wide range of algorithms to suit different scenarios.

Authentication algorithms

The integrity of messages can be checked with many different algorithms. These include:

- HMAC-MD5
- HMAC-SHA1
- HMAC-SHA2

- AEAD

To prove the integrity of the data being sent, the sender runs the information through a hash function to return a unique string of characters. These are special formulas that will always return the same result whenever they receive the same input.

The sender signs this data with their private key to form what is known as a digital signature. The digital signature is then attached to the message and sent to the recipient. To check whether the data retains its integrity and hasn't been tampered with, **the recipient decrypts the hash with the sender's public key**. They then use the same hash function on the data that was sent. The recipient then compares the two values.

If they are the same, it means that the data has not been altered since it was signed by the sender. If they are different, it is likely that the data has been tampered with, or there has been some other error.

That's the short version of how hash functions can be used to show the integrity of data. If you would like a more in-depth understanding, check out our article on encryption, salting and hashing.

Encryption algorithms

TLS uses symmetric-key encryption to provide confidentiality to the data that it transmits. Unlike public-key encryption, just one key is used in both the encryption and decryption processes. Once data has been encrypted with an algorithm, it will appear as a jumble of ciphertext. As long as an appropriate algorithm is used, attackers will not be able to access the actual data, even if they intercept it.

TLS can use many different algorithms, such as Camellia or ARIA, although the most popular is the Advanced Encryption Standard (AES).

Compression

Compression is the process of encoding data to make it take up less room. TLS supports compression if both sides of the connection decide to use it. Despite this ability, it is generally recommended to avoid using TLS to compress data, especially since the CRIME attack (see the **TLS Security Issues** section below) was found to be able to take advantage of compressed data for session hijacking.

Padding

Padding adds extra data to a message before it is encrypted, so that blocks of data the same size are sent. It's a common cryptographic process that is used to help prevent hints in the structure of encrypted data from giving away its true meaning. TLS generally applies PKCS#7 padding to records before they are encrypted.

Alert protocol

If the connection or security becomes unstable, compromised, or a serious error has occurred, **the alert protocol allows the sender to notify the other party**. These messages have two types, either warning or fatal. A warning message indicates that the session is unstable and allows the recipient to determine whether or not the session should be continued.

A fatal message tells the recipient that the connection has been compromised or a serious error has occurred. The sender should close the connection after they send the message. The alert protocol also contains information about what is causing the particular connection problem. This can include things like decryption failure, an unknown certificate authority, an illegal parameter and much more.