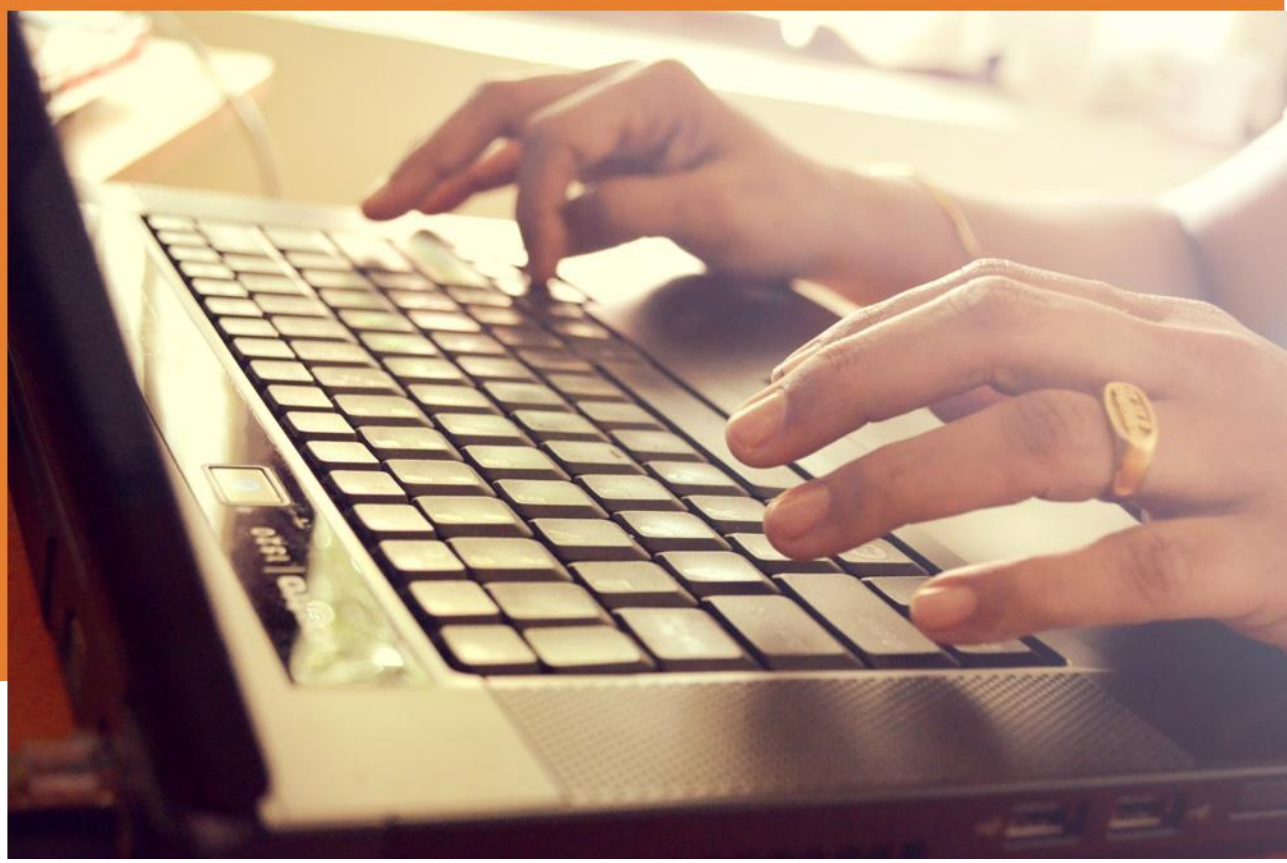
SEPTEMBER 2015

Secure Network Protocols

How SSL/TLS, SSH, SFTP and FTPS work



Bruce P Blackshaw

EnterpriseDT
SECURE FILE TRANSFER & SHARING

Executive Summary

Network security is an important topic in a world where most corporations have an on-line presence and billions of dollars of e-commerce is transacted daily. Technologists need to have an understanding of the basic concepts underlying secure networks and the network protocols that they use.

This eBook explains the fundamentals of encryption, and how the two most widely used secure network protocols operate – SSL/TLS and SSH. It also examines in detail two important file transfer protocols that are implemented using SSL/TLS and SSH – FTPS and SFTP respectively.

Finally, FTPS and SFTP are compared as to their suitability in the enterprise. The most important criteria is the existing infrastructure – if there is an existing significant investment in one of these technologies then this will normally override technical considerations. However on a feature comparison basis, SFTP is a superior solution for securely transferring files.

Bruce Blackshaw has been writing software professionally for almost 25 years. He has wide experience in encryption, security, and network protocols such as SSL/TLS, SSH, SFTP and FTPS across a variety of industries. Bruce is a founding partner of Enterprise DT and is currently one of the principal developers of their flagship product for secure and reliable file transfer, CompleteFTP.

www.enterprisedt.com | sales@enterprisedt.com
PO Box 3027, Yeronga QLD 4104, AUSTRALIA | +61-7-3053 8544

Table of Contents

Introduction

This ebook explores how secure network protocols work. It will explain key concepts such as encryption, cryptographic hashes and public key encryption. The two most popular secure network protocols, SSL/TLS and SSH, will be examined, and their secure file transfer counterparts, FTPS and SFTP will be described and compared.

What is Encryption?

Encryption is the process of encoding information in such a way that only parties who are authorized to read the encrypted information are able to read it. Its goal is to keep information secure from eavesdroppers, or secret.

The unencrypted information is known as the plain-text, while the encrypted information is called the cipher-text. To obtain the plain-text from the cipher-text, an encryption key is required, and only authorized parties have a copy of the encryption key. The encoding process is known as the encryption algorithm. The algorithm is designed such that decrypting the plain-text without the key is not practically possible.

There are two main types of encryption – *symmetric key encryption* and *asymmetric, or public key encryption*.

Symmetric Key Encryption

In symmetric encryption, the key used to encrypt the plain-text and the key used to decrypt the cipher-text is the same. This means that the two parties (the sender and receiver) must share the key (which itself must be kept secret). Of course, working out how to share the key securely is another instance of what encryption is designed for – sharing information securely. So how do the two parties share their secret key? Fortunately, this can be achieved by asymmetric (or public key) encryption, explained below. Popular symmetric key algorithms include AES, Blowfish, RC4 and 3DES.

Public key encryption

Public key encryption is based on a special set of algorithms that require two separate keys. One key, known as the private key, is kept secret, and the other key, the public key, is made widely available. Together they are known as the key-pair. Generally, anyone can use the public key for encryption, but only the owner of the private key can decrypt it.

The advantage of a public key encryption system is this: secret (i.e. encrypted) messages can be sent to anyone who has published their public key, and only the recipient will be able to decrypt the message. So as long as their public key can be trusted to be theirs (an important caveat!), a secure system for exchanging secret messages can easily be set up. Each party can publish their public key and send secret messages to the other using the other's public key. They use their own private key to decrypt messages that they receive.

But doesn't publishing the public key make encrypted messages more vulnerable to unauthorized decryption? No, it is not practically possible to derive the private key of a key-pair from the public key, and without the private key, the cipher-text cannot be decrypted. So publishing the public key does not make it easier to decrypt messages encrypted by the public key.

RSA and Diffie–Hellman were the earliest public key algorithms. For a long time it was thought they were invented in 1976/1977, but when secret GCHQ research was declassified in 1997, it turned out they had been independently conceived of a few years earlier. ElGamal and DSS are other well-known public key algorithms.

There are a number of important uses of public key encryption described below.

Key distribution

Symmetric encryption uses a single secret key that both parties require, and ensuring that this secret key is securely communicated to the other party is difficult. This is known as the key distribution problem.

Public key encryption is ideally suited to solve this problem. The receiving party, who requires the sender's secret symmetric key, generates a key-pair and publishes the public key. The sender uses the receiver's public key to encrypt their symmetric key, and sends it to the receiver. Now, both sender and receiver have the same secret symmetric key, and no-one else does as it has never been transmitted

as clear-text. This is often known as the key exchange.

An obvious question is to ask why not use public key encryption for everything, and avoid having to send a secret key altogether? It turns out that symmetric encryption is orders of magnitude faster at encryption and decryption. So it is much more efficient to use public key encryption to distribute the symmetric key, and then to use symmetric encryption.

Digital Signatures

Public key encryption is an important component of digital signatures. A message can be signed (encrypted) with a user's private key, and anyone can use their public key to verify that the user signed the message, and that the message was not tampered with. This application of public key encryption is explained in more detail in the next section, Cryptographic hashes.

Certificate authorities

A critical requirement for a system using public key encryption is providing a way of reliably associating public keys with their owners. There is limited value in being able to use someone's public key to encrypt a message intended for them if it can't be determined that it really is their public key. This is what certificate authorities are for, and both they and certificates are explained below.

Cryptographic hashes

Cryptographic hash algorithms are important mathematical functions used widely in software, particularly in secure protocols such as SSL/TLS and SSH.

A block of data is passed through a hash algorithm to produce a much smaller hash value, known as the *message digest*, or simply the *digest*. The same message will always result in the same digest. Different messages produce different digests.

An important feature of hash algorithms is that given a particular digest, it is extremely difficult to generate a message that will produce it. They are “one way" algorithms – the digest of a message is easy to calculate, but a message can't be deduced from the digest. It is mathematically possible to have two different messages produce the same digest – known as a collision – but for good hash algorithms this is extremely unlikely.

Popular hash algorithms include MD5 and SHA-1, although these are now being phased out in favour of stronger algorithms such as SHA-2.

Hash algorithms are used for many purposes, such as verifying the integrity of data or files, password verification, and building other cryptographic functions such as message authentication codes (MACs) and digital signatures.

Digital signatures

A written signature demonstrates that a document was created by a known author and accurately represents them. A digital signature is similar – it guarantees that the message was created by a known sender (authentication) and that the message was not tampered with in transit (integrity).

To sign a message requires two stages. Firstly, the message digest is calculated, producing a unique hash that is typically much smaller than the message. Next, the digest is encrypted using the message signer's private key. This is the digital signature of the message.

To verify the signer of a message also requires two stages. Firstly, the signer's public key (which is widely available) is used to decrypt the digital signature, yielding the message digest. Then the message digest of the message is calculated and compared to the decrypted digest. If the message has not been tampered with, the digests should be identical. And because the signer's public key was used to decrypt the signature, the signer's private key must have been used to encrypt it.

Why use the message's digest at all? Why not just encrypt the message with the signer's private key and use the encrypted message as the signature? While that would certainly work, it is impractical – it would double the size of the message when the signature is included. The digest is very small and of a fixed size, so encrypting the digest produces a signature that is much smaller.

Message authentication codes (MAC)

A message authentication code, or MAC, is a small piece of information attached to a message that can verify that the message has not been tampered with, and authenticate who created it.

A special type of MAC is the HMAC, which is constructed using a cryptographic hash and a secret key. The secret key is padded and concatenated with the message, and the digest, or hash, is calculated. This digest is then concatenated again with the padded secret key to yield the HMAC value. It is impossible for an attacker to produce the same HMAC without having the secret key.

The sender and receiver both share the secret key. When the receiver gets a message, they calculate the HMAC and compare it to the HMAC provided with the message. If the HMACs match, only someone possessing the secret key could have produced the message. The secret key itself is never transmitted.

Passwords, Password Hashes and Salts

Cryptographic hashes are extremely useful for systems that require password verification. It is an unjustifiable security risk to store user's passwords, even if they are encrypted. Instead, the digest of each password is stored. When the user supplies the password, it is hashed and compared with the digest that is stored. This is preferable because the password cannot be recovered from its hash.

One drawback with this method is that if users have the same password, they will have the same hash value. Tables of pre-calculated digests for common passwords can be used to attack a system if the file containing the digests can be obtained. These tables are known as rainbow tables.

For this reason a salt – a random, non-secret value – is concatenated with the password before the digest is calculated. Because every user has a different salt, it is not feasible to use pre-calculated tables – there would need to be a table for every possible salt value. For salts to be effective, they must be as random as possible, and of adequate size – preferably at least 32 bits.

What are Certificates?

While discussing public key encryption, it was explained that there needs to be a way of reliably associating public keys with their owners. Using someone's public key to encrypt a message intended for them requires knowing that it is indeed their public key.

Certificate Authorities are the solution to this problem. A Certificate Authority (a "CA") is an organization that issues digital certificates. A digital certificate is an electronic document that certifies ownership of a public key.

A digital certificate contains a number of fields – the public key that it is certifying ownership of, the name of the owner (the subject), the issuer name (i.e. the CA), the start and end dates, and the issuer's digital signature. The digital signature verifies that the CA actually issued the certificate. Digital signatures are explained in more detail here.

For the system to work, the certificate authority must be a trusted third party. There are only a small number of CAs, including Comodo, Symantec and GoDaddy. CAs issue their own certificates containing their public keys, which are known as trusted root certificates.

To obtain a certificate from a CA, an organization must supply the CA with its public key, and sufficient documentation to establish that it is a genuine organization. The CA verifies these details before issuing the certificate.

Website validation

The most common use of certificates is to validate HTTPS websites (i.e. websites that have a URL beginning with https://). When a web browser connects to a site such as Amazon, the user needs to know that the site can be trusted, i.e. that the URL www.amazon.com actually refers to a site controlled by the company called Amazon. This is done by embedding the website domain name in the certificate's subject field when applying to a CA for the certificate. The CA ensures that the domain name is controlled by the organization before issuing the certificate. The web browser has its own list of root certificates, and when it connects to the site, the site's certificate is sent back by the web server. Using the CA certificate, it checks that the certificate sent by the web server was issued by one of the CA's it recognizes and that the domain name matches the domain name in the certificate.

Why is this check important? As long as Amazon owns its domain name (which we know it does), why do we need the browser to check the certificate?

Unfortunately, it is possible for malicious software to impersonate another machine. When a URL is entered into a web-browser, such as https://www.amazon.com, it must be translated to an IP address, e.g. 192.168.1.64. These digits are what the browser uses to connect to the web-server. The process of translation is called a DNS lookup, and it involves checking the public register of domain names to get the IP address Amazon has decided to use. Malicious software can compromise DNS lookups, returning the wrong IP address, which might be for a fake website that looks similar to Amazon and is designed to obtain credit card details.

This is where the certificate check proves its worth – the fake website can't return the genuine certificate, and the web-browser will signal that the certificate returned is not registered to the domain name used in the URL. In most browsers the genuine site will display a padlock symbol, and clicking on it with a mouse will show the site's verified identity, as with Chrome, below.

This is why it is important to use URLs that begin with https rather than http – via the certificate, the browser can provide an assurance that the site being connected to is a verified owner of the domain.

How does SSL/TLS work?

History

The Secure Sockets Layer (SSL) is a cryptographic protocol designed to secure communications over TCP/IP networks. SSL was developed by Netscape during the early 1990's, but various security flaws meant that it wasn't until SSL 3.0 was released in 1996 that SSL became popular.

It was also during this time that an open source implementation of SSL called SSLeay was made available by Eric Young, which helped ensure its widespread adoption on the Internet. The Apache web server was also gaining in popularity, and Ben Laurie of Apache fame used SSLeay to produce Apache-SSL, one of the first freely available secure web servers.

SSL became Transport Layer Security (TLS) with the publication of the TLS 1.0 standard in 1999, followed by TLS 1.1 and TLS 1.2, the most recent version. All versions of TLS are in widespread use, and it is only recently that support for SSL 3.0 has been discontinued in response to the POODLE vulnerability. For simplicity, we'll refer to SSL/TLS as TLS for the remainder of this article.

Overview

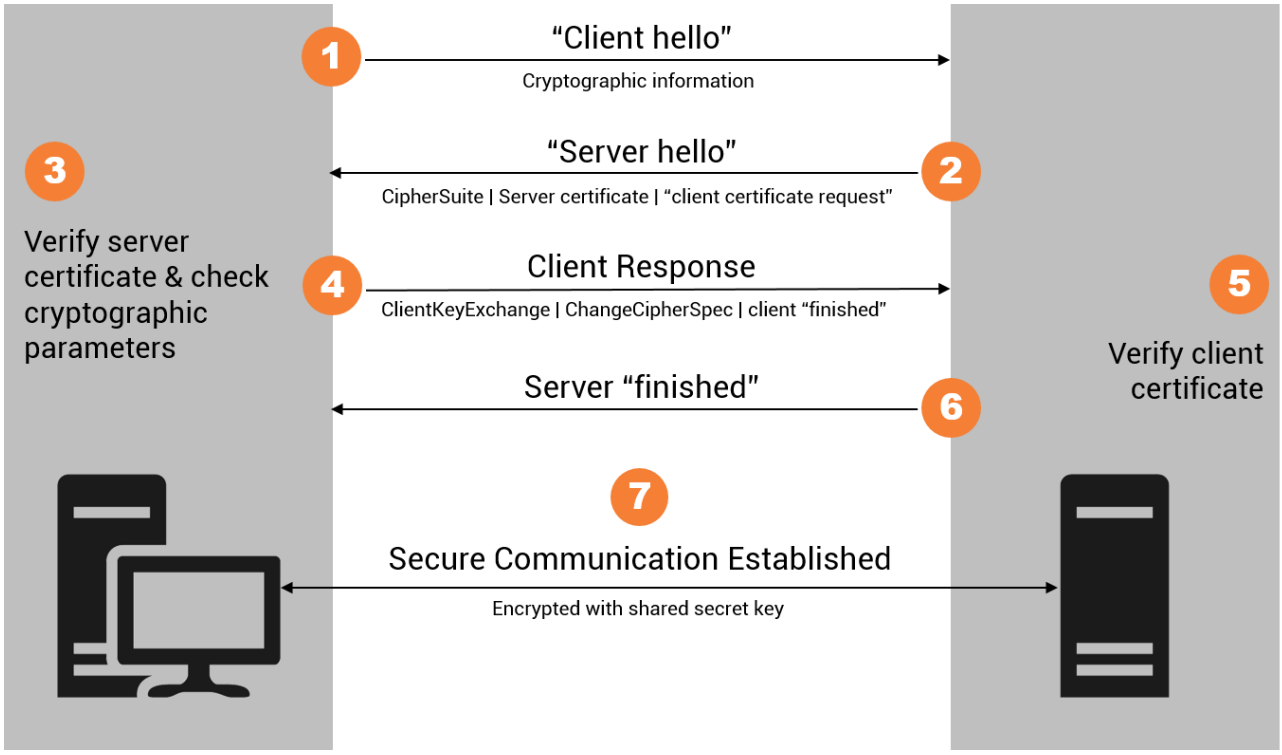
TLS is intended to provide secure connections between a client (e.g. a web browser), and a server (e.g. a web server) by encrypting all data that is passed between them.

Ordinary network connections are not encrypted, so anyone with access to the network can sniff packets, reading user names, passwords, credit card details and other confidential data sent across the network – an obviously unacceptable situation for any kind of Internet-based e-commerce.

Earlier in this white paper we have discussed how encryption works, including public key encryption and certificates. TLS uses public key encryption to verify the parties in the encrypted session, and to provide a way for client and server to agree on a shared symmetric encryption key.

The SSL Handshake

The “handshake" is the most critical part of SSL/TLS, as this is where the important parameters for the connection are established. The various steps in the handshake are described below.



Step 1 – client hello

After establishing a TCP/IP connection, the client sends a ClientHello message to the server. This states the maximum TLS version the client is willing to support, a random number, the list of cipher suites supported in order of preference, and the compression algorithms. Cipher suites are identifiers for a group of cryptographic algorithms that are used together. For example, TLS_RSA_WITH_AES_128_CBC_SHA means that the server's RSA public key is to be used, and the encryption algorithm is 128 bit AES. The Message authentication codes (MAC) algorithm is HMAC/SHA-1.

The ClientHello is sent in cleartext, so anyone able to intercept the network packets can read it.

Step 2 – server hello

The server replies to the ClientHello with a ServerHello message. It tells the client the TLS version to use, together with the cipher suite and compression algorithm it has chosen. The ServerHello also provides a server-generated random number and a session identifier. The ServerHello is also sent in cleartext.

Immediately after sending its ServerHello, the server sends its certificate, containing its public key, to

the client. This is followed by an optional ServerKeyExchange message which contains any additional required values for the key exchange.

If the server is configured to require the client to identify itself with a client certificate, the server asks for it at this point in the handshake via the optional CertificateRequest message.

Finally, the server sends the client a ServerHelloDone message, still in cleartext.

3 Step 3 – verify server certificates

Typically, the client has a cache of certificates or CA root certificates by which it can verify that the server's certificate is genuine (and registered to the server's IP address). If the server's certificate is unknown, the client may give the option of accepting the certificate anyway (which is potentially dangerous), or may sever the connection. This message is sent in cleartext.

4 Step 4 – client response

If the server requested a certificate from the client, the client sends its certificate, followed by the ClientKeyExchange message.

For the ClientKeyExchange message, the client generates what is called the premaster secret, consisting of 48 bytes. This secret is sent to the server as part of this message, but is encrypted with the server's public key (obtained from the server's certificate) so that only the server can decrypt it with its private key (as messages are still being sent as plain text).

Once the client and server share the premaster secret, they each use it in combination with both of the random values that have been exchanged earlier to produce the master secret and subsequently the session keys – the symmetric keys used to encrypt and decrypt data in the subsequent TLS session.

The ChangeCipherSpec message is sent after the ClientKeyExchange message. This message indicates to the server that all subsequent messages will be encrypted using the newly created session keys. It is followed by the Finished message, the first to be encrypted. The Finished message is a hash of the entire handshake so far that enables the server to verify that this was the client that has been communicating with the server throughout the handshake.

5 Step 5 – verify client certificate

If the server requested the client's certificate, it is verified to ensure it is correct.

Step 6 – server finished

The server replies to the Finished message from the client with a ChangeCipherSpec message of its own, followed by an encrypted Finished message, which again is a hash of the handshake to this point. This enables the client to verify that this is the same server that has been communicating with it during the handshake.

Step 7 – secure communication established

By this point, all messages are encrypted and so a secure communication channel across the network between client and server has been established.

Records and alerts

How is data packaged up and sent across the network after the handshake is completed? This involves the *record protocol* and the *alert protocol*.

Record protocol

The record protocol is responsible for compression, encryption and verification of the data. All data to be transmitted is split into records. Each record consists of a header byte, followed by the protocol version, the length of the data to be sent (known as the payload), and the payload itself. Firstly, the data is compressed if compression has been agreed upon. Then a MAC is computed and appended to the data. The MAC allows the receiver to verify that the record has not been tampered with. Its calculation includes a sequence number which sender and receiver both keep track of. Finally, the data and the appended MAC are encrypted using the session's encryption keys, and the result is the payload for the record. At the receiving end, the record is decrypted, and the MAC is calculated to verify that the record's data has not been tampered with. If compression was used, it is decompressed.

Alerts

If errors occur, SSL/TLS defines an alert protocol so that error messages can be passed between client and server. There are two levels – warning and fatal. If a fatal error occurs, after sending the alert the connection is closed. If the alert is a warning, it is up to the party receiving the alert as to whether the session should be continued. One important alert is close notify. Sent when either party decides to close the session, close notify is required for normal termination of a session. It is worth noting that some SSL/TLS implementations do not send this message – they simply terminate the connection.

Versions

SSL/TLS internal version numbers do not correspond as might be expected to what is publicly referred to as the version number. For example, 3.1 corresponds to TLS 1.0. The main versions currently in use are shown here.

Major Version	Minor Version	Version Type
3	0	SSL 3.0
3	1	TLS 1.0
3	2	TLS 1.1
3	3	TLS 1.2

These are useful to be familiar with, as the internal version numbers are often preferred.

SSL/TLS vulnerabilities

TLS is a mature, widely used secure network protocol that will be securing transactions on the Internet for many years to come. Like any secure protocol however, a number of important vulnerabilities have been discovered over the years. Vulnerabilities will continue to be discovered, and it is important to keep software that utilises TLS up-to-date so that the latest security patches are applied.

Some of the more well-known vulnerabilities and how they have been addressed is discussed below.

Heartbleed

Heartbleed is one of the most serious vulnerabilities ever found in TLS software, allowing the theft of server keys, user session IDs and user passwords from compromised systems. It was not, however, an SSL protocol flaw, but rather an implementation bug (known as a buffer over-read) in OpenSSL's free library, which is widely used across the Internet. Millions of machines were affected, and numerous successful attacks reported.

Software systems not using the relevant versions of OpenSSL were not affected. OpenSSL was rapidly patched, but patching millions of machines takes time. Not only did machines need to be patched, but server private keys must be updated, user passwords changed and certificates re-issued. A year later, it is likely that there are still compromised machines on the Internet that have not been suitably modified.

The total cost of Heartbleed has been estimated to be in the range of hundreds of millions of dollars.

POODLE

POODLE is a vulnerability in an older SSL protocol, SSL 3.0. While most systems use TLS 1.0, 1.1 or 1.2, the TLS protocol has a fall-back provision to allow interoperability with older software still using SSL 3.0. So POODLE attacks use this fall-back provision to fool servers into downgrading to SSL 3.0.

The simplest fix is to disable SSL 3.0 in clients and servers. SSL 3.0 was published in 1996, it has long been superseded, and there should be no need to support it after almost 20 years. POODLE is a far less serious vulnerability than Heartbleed.

RC4

RC4 is a widely used TLS cipher that is no longer regarded as secure. RC4 is also known as ARC4 or ARCFOUR (because RC4 is trademarked). Its speed and simplicity made RC4 popular, but recently (February 2015) RFC7465 recommended that it no longer be used.

SSL/TLS File Transfer Protocol: FTPS

One of the most common uses of SSL/TLS is a secure form of file transfer known as *FTPS*.

Traditional FTP as defined in RFC 959 makes no mention of security. This is understandable as it was written in 1985 and based on specifications from the 1970s. This was when universities and the military were the primary users of the Internet and security was not the concern that it is today.

As a result, in FTP user-names and passwords are (still) sent over the network in clear text, meaning anyone able to sniff the TCP/IP packets is able to capture them. If the FTP server is on the Internet, the

packets pass through public networks, and should be considered to be publicly available.

It was not until the 1990s when Netscape developed their Secure Sockets Layer (SSL) that a solution became practical. A draft RFC in 1996 described an extension to FTP called FTPS that allowed FTP commands to be used over an SSL connection, and by 2005 this was developed into a formal RFC.

Implemented by clients such as Filezilla and servers such as ProFTPD, FTPS quickly became popular.

Implicit FTPS

There are two forms of FTPS – implicit mode and explicit mode. Implicit mode FTPS is obsolete and not widely used, but is still occasionally encountered.

Implicit FTPS does not have an explicit command to secure the network connection – instead it does so implicitly. In this mode, the FTPS server expects the FTPS client to immediately initiate an SSL/TLS handshake upon connecting. If it does not, the connection is dropped. The standard server port for implicit mode connections is 990 (not the standard port 21 used for FTP).

Once the SSL/TLS connection is established, the standard FTP commands are used to navigate the server's file system and to transfer files. As the connection is secure, passwords can be sent and data cannot be inspected by eavesdroppers.

Explicit FTPS

In explicit FTPS mode, the client must explicitly request the connection to be secured by sending the AUTH TLS command to the server. Once this command is sent the SSL/TLS handshake commences as with implicit TLS, and the command connection is secured.

The advantage of using explicit mode FTPS over implicit mode is that the same port number as standard FTP can be used – port 21. Ordinary FTP users simply do not send the AUTH command, and so they never secure the connection. The server administrator can optionally require the AUTH command to be used if they do not wish unsecured file transfers to be made.

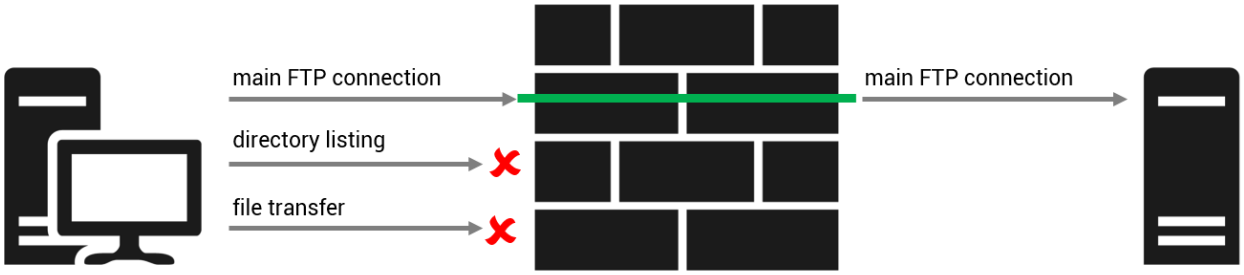
Explicit mode FTPS should always be used in preference to implicit mode, primarily because implicit mode has been deprecated for many years.

Disadvantage of FTPS

FTPS has one significant disadvantage, which is its use of a separate network connection for data, including file contents and directory listings. This is actually part of the FTP protocol – commands are sent via the initial "control" connection on port 21, and whenever data is transferred, a new network connection must be established for the transfer. The client and server must agree on a port number, and a connection must be opened.

With unencrypted FTP, this isn't too problematic. There can be issues with an exhaustion of network connections if too many transfers are made within a short period of time. As each transfer requires a new connection, and operating systems usually require a few minutes to free up closed connections, many transfers of small files can result in eventual errors.

The more significant problem is getting through firewalls. Firewalls are normally configured to allow access via port 21. Modern firewalls are also clever enough to be able to inspect the commands sent between client and server (PORT or PASV) to be able to determine which ports must be dynamically opened to allow data transfers.



With FTPS, however, the commands are on an encrypted channel, and firewalls cannot inspect them. This means they cannot automatically open data ports, and so transfers and directory listings fail. Instead, a fixed range of ports must be agreed in advance, and configured in client, server and firewall.

Future of FTPS

Nowadays, FTPS has a strong competitor in SFTP, or SSH File Transfer Protocol. They are completely different protocols, and their relative merits will be examined in the next section. However, FTP and FTPS have a huge install-base and will no doubt continue to be widely used for many years to come.

How does SSH work?

SSH History

In the late 1980's and 1990's, network tools such as rlogin and telnet were commonly used for logins into remote machines, typically on Unix platforms. These tools allowed users to open command shells that permitted them to execute commands on the remote machines as if they were actually on the machine, and were extremely useful for systems administration.

There was one critical drawback – none of these tools were secure. Passwords were sent over networks in plain-text, meaning anyone able to sniff the network could obtain credentials for the remote machine. This problem is why Tatu Ylönen, a Finnish researcher at the Helsinki University of Technology, decided a secure network protocol was required. In 1995 he wrote the first version of SSH, known as SSH-1, and released it as freeware. It consisted of a secure server and client.

As its popularity grew rapidly, Ylönen founded SSH Communications Security to market and develop SSH as a proprietary product. In 1999 Björn Grönvall began working on an earlier freeware version, and the OpenBSD team funded his work to produce the freely available OpenSSH. Ports were soon made to many other platforms, and OpenSSH remains the most widely known and used version of SSH.

In 2006 SSH 2.0 was defined in RFC 4253. SSH-2 is incompatible with SSH-1, and has improved security and features, rendering SSH-1 obsolete.

SSH overview

SSH is a secure network protocol that can be used on any platform for any purpose requiring secure network communication. Typical uses include:

- secure remote login tools, such as the ssh client;
- secure file transfer, such as the scp and sftp tools; and
- secure port forwarding or secure tunnelling.

SSH-2 uses a layered architecture, and consists of a transport layer, a user authentication layer, and a connection layer.

The transport layer runs over TCP/IP, and provides encryption, server authentication, data integrity

protection, and optional compression. The user authentication layer handles client authentication, while the connection layer provides services such as interactive logins, remote commands, and forwarded network connections.

The transport layer

The transport layer is message-based, and provides encryption, host authentication and integrity checking. Messages are sent between client and server over TCP/IP via the binary packet protocol – "packets" of data are exchanged in the format defined below, and the payload of each packet is the message:

```
uint32 packet_length
byte padding_length
byte[n1] payload; n1 = packet_length - padding_length - 1
byte[n2] random padding; n2 = padding_length
byte[m] mac (Message Authentication Code - MAC); m = mac_length
```

The MAC is an important field, because it is the MAC that allows recipients of messages to be sure that messages have not been tampered with – the integrity checking referred to above. The MAC is calculated over the rest of the data in the packet, and uses a shared secret established between client and server, and a sequence number which both parties keep track of. MACs are described in this post.

Establishing a Session

How does a session between a client and a server begin? Each side sends an identification string once the TCP/IP connection has been established. This string is in the following format:

```
SSH-2.0-softwareversion SP comments CR LF
```

Here "SP" means a space, "CR" is a carriage return character, and "LF" is a line feed character. The software version is the vendor version, and the comments and space are optional. So in the case of CompleteFTP, when a client connects to the server they receive the following string:

```
SSH-2.0-CompleteFTP_9.0.0
```

The string end with the mandatory carriage return and line feed – no comments are used.

Once identification strings are exchanged, a number of options must be agreed upon – the ciphers used for encryption, the MAC algorithms used for data integrity, the key exchange methods used to set up one-time session keys for encryption, the public key algorithms that are used for authentication, and finally what compression algorithms are to be used. Both client and server send each other an SSH_MSG_KEXINIT message listing their preferences for these options:

```
byte         SSH_MSG_KEXINIT
byte[16]     cookie (random bytes)
name-list    kex_algorithms
name-list    server_host_key_algorithms
name-list    encryption_algorithms_client_to_server
name-list    encryption_algorithms_server_to_client
name-list    mac_algorithms_client_to_server
name-list    mac_algorithms_server_to_client
name-list    compression_algorithms_client_to_server
name-list    compression_algorithms_server_to_client
name-list    languages_client_to_server
name-list    languages_server_to_client
boolean      first_kex_packet_follows
uint32       0 (reserved for future extension)
```

This message is the payload of a binary protocol packet whose format is described above. Name lists of algorithms are comma-separated. The client sends algorithm lists in order of preference, while the server sends a list of algorithms that it supports. The first supported algorithm in order of the client's preference is the algorithm that is chosen. Given both messages, each side can work out what algorithms are to be used.

After SSH_MSG_KEXINIT, the selected key exchange algorithm, which may result in a number of messages being exchanged. The end result is two values: a shared secret, K, and an exchange hash, H. These are used to derive encryption and authentication keys. An SSH_MSG_NEWKEYS is sent to signify the end of these negotiations, and every subsequent message uses the new encryption keys and algorithms.

With the SSH-2 connection established, the client requests a “service” (usually ssh-userauth to begin the authentication process) with the SSH_MSG_SERVICE_REQUEST message.

Authentication layer

The next step is for the client to identify itself to the server, and be authenticated. This is managed via the user authentication layer, which runs on top of the transport layer. This means user authentication messages are encrypted and exchanged using the transport layer.

User authentication is initiated by the client with a "service" request for the ssh-userauth service. If the server responds by allowing the request, the client sends an authentication request, which includes their username and the authentication method.

There are a number of possible authentication methods, and which one is used will depend on the client and server's support for it. The most popular method is password authentication, which is self-explanatory. Another is publickey authentication. Typically, the client proposes a method, and the server either accepts or rejects that method.

An example of a password authentication request by a user called "enterprisedt" is shown below:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    "enterprisedt" [user name]
string    "ssh-userauth" [service name]
string    "password"  [authentication method]
boolean   FALSE
string    "mypassword" [user's password]
```

The server will validate the password sent for this user against the details it has stored for the user. The server will not store the user's actual password for validation, but a cryptograpic hash of the password, which cannot be reverse-engineered to obtain the password.

If the user authentication request is rejected (for example, an incorrect password was supplied), a failure message is sent by the server, and it provides a list of alternative authentications that can be tried.

It is common to send "none" as the initial authentication method, and the server will usually respond with a failure message containing a list of all available authentication methods. An example response to "none" is shown below:

```
byte          SSH_MSG_USERAUTH_FAILURE
name-list     password,publickey
boolean       FALSE (partial success flag)
```

Here the server is informing the client that either password or publickey authentication can be used.

If the password authentication request succeeds, the server returns a success message as shown below:

```
byte      SSH_MSG_USERAUTH_SUCCESS
```

At this point authentication is complete, and other services can be requested. These can include TCP/IP forwarding requests, and channels for terminal access, process execution and subsystems such as SFTP.

Connection layer

The final piece of SSH-2's layered architecture is the connection layer, which provides network services such as interactive sessions and port forwarding on top of the transport layer, which supplies the necessary security.

Once established, an SSH connection can host one or more SSH channels, which are logical data pipes multiplexed over the connection. The client can open multiple channels on the one connection to the same server, and perform different network tasks on different channels. In practice, SSH implementations rarely use multiple channels on a connection, preferring to open a new connection for each channel.

An important feature of SSH channels is flow control. Data may only be sent across a channel when the recipient has indicated they are ready to receive it – a form of sliding-window flow control. The size of the window is established by the recipient when the channel is opened, and the window size is decremented as data is sent. Periodically, the recipient sends a message to increase the window size.

The SSH_MSG_CHANNEL_OPEN message used to open an interactive session is shown below. This session might be subsequently used for a terminal session, to run a remote command, or to start a subsystem such as SFTP.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "session"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
```

The initial window size sets the number of bytes the recipient of this message can send to the sender, while the maximum packet size is the largest amount of data that it will accept in a single message.

The recipient of this message replies with an SSH_MSG_CHANNEL_OPEN_CONFIRMATION message if it is prepared to open the requested channel:

```
byte      SSH_MSG_CHANNEL_OPEN_CONFIRMATION
uint32    recipient channel
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
```

Once a channel has been successfully opened, data can be exchanged, and channel-specific requests can sent. When the sliding-window size for either the client or server becomes too small, the owner of the window sends a SSH_MSG_CHANNEL_WINDOW_ADJUST message to increase it:

```
byte      SSH_MSG_CHANNEL_WINDOW_ADJUST
uint32    recipient channel
uint32    bytes to add
```

Data is sent across the channel via the SSH_MSG_CHANNEL_DATA message. How the data is used will depend on the type of channel that has been established:

```
byte      SSH_MSG_CHANNEL_DATA
uint32    recipient channel
string    data
```

Channel requests are used to perform particular actions over a channel. Common requests include starting a shell or executing a remote command. For example, a remote shell is started by the request shown below:

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "shell"
boolean   want reply
```

A remote command is executed by the following request:

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "exec"
boolean   want reply
string    command
```

Finally, an SFTP subsystem can be opened by this request:

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "subsystem"
boolean   want reply
string    "sftp-server"
```

Subsystems are sets of remote commands that are pre-defined on the server machine. The most common is SFTP, which provides commands to transfer and manipulate files. The subsystem commands (including the SFTP protocol) run over SSH, i.e. data for the subsystem commands is sent in SSH_MSG_CHANNEL_DATA messages. When one of these messages arrives at the client or server, it is passed to the subsystem for processing.

Once either the client or server has finished using the channel, it must be closed. The SSH_MSG_CHANNEL_EOF message is sent to indicate no more data will be sent in the direction of this message. The SSH_MSG_CHANNEL_CLOSE message indicates the channel is now closed. The recipient must reply with an SSH_MSG_CHANNEL_CLOSE if they have not already sent one. Once closed, the channel cannot be re-opened.

SSH File Transfer Protocol: SFTP

The most common subsystem used with SSH is SFTP, which provides commands to transfer and manipulate files. SFTP is also known as the SSH File Transfer Protocol, and is a competitor to FTPS – traditional FTP over an SSL/TLS connection.

The SFTP subsystem runs over the SSH transport layer, but it is a sophisticated message-based protocol in its own right. SFTP messages are transmitted as the data field of the transport layer's SSH_MSG_CHANNEL_DATA message

SFTP messages are in a standard format, as shown below:

```
uint32          length
byte            type
uint32          request-id
... type specific fields …
```

The first field is the length of the message, excluding the length field, and next is the message type. The third field is the request id – every request sent from the client has a request id, and the server's reply must include the corresponding request id.

Some of the more important SFTP messages together with their type ids are shown and described below:

```
SSH_FXP_INIT               1
SSH_FXP_VERSION            2
SSH_FXP_OPEN               3
SSH_FXP_CLOSE              4
SSH_FXP_READ               5
SSH_FXP_WRITE              6
SSH_FXP_STATUS           101
SSH_FXP_DATA             103
```

SSH_FXP_INIT is the first message sent by the client to initiate the SFTP session, and the server replies with SSH_FXP_VERSION, indicating the versions it supports.

SSH_FXP_OPEN requests the server to open a file (or optionally create it if it does not exist), while

SSH_FXP_CLOSE closes a file. SSH_FXP_READ asks to read a certain byte range from a file, and the server responds with SSH_FXP_DATA, which contains the requested bytes. SSH_FXP_WRITE is used to write data to a file, and one of its fields is the data to write (as well as the offset into the file).

If any of the above commands fail, SSH_FXP_STATUS is returned with an error code indicating the type of error that occurred. It is also used to signal a successful write in response to SSH_FXP_WRITE.

There are also commands for other standard file and directory operations, such as removing files (SSH_FXP_REMOVE), renaming files (SSH_FXP_RENAME), and creating and removing directories (SSH_FXP_MKDIR, SSH_FXP_RMDIR). Directories are read by opening them (SSH_FXP_OPENDIR) and sending SSH_FXP_READDIR requests. Again, SSH_FXP_STATUS is used to indicate success or failure of these requests.

There is no specific SFTP message to terminate an SFTP session – instead, the client closes the SSH channel being used.

It is important to note that SFTP is an entirely different protocol to traditional FTP, i.e. it is not FTP commands sent over an SSH connection. By contrast, FTPS is FTP commands sent over an SSL/TLS connection. The two protocols are easily confused, as they are both secure protocols for transferring files.

SFTP vs FTPS

We've described how FTPS and SFTP work, above. Essentially, both protocols achieve exactly the same thing – secure file transfer and secure, remote manipulation of file-systems.

They are, however, completely different protocols, and people implementing a secure file transfer solution will need to decide which protocol to use.

Existing usage is naturally an important consideration. If SFTP and/or SSH is already used in other areas of an organization, it is prudent to use SFTP. Existing knowledge and skills within the organization can be leveraged, as well as technical infrastructure. Similarly, if FTP and/or FTPS is already used elsewhere, it may be best to use FTPS.

Project requirements may also dictate the protocol. If a server-side solution is being implemented, it may be that clients are restricted to a particular protocol, and so no decision need be made.

But what if the starting point is a completely clean slate and there are no constraints on which protocol that could be used? Is there a clear winner?

SFTP a clear winner

Yes, and it is SFTP. A few years ago, such a decision was not as straightforward, mainly because of the dominance of the FTP protocol in most organizations. Now clients and server software is widely available for both SFTP and FTPS – in fact many applications such as CompleteFTP support both.

This means a decision can be made on purely technical grounds, and SFTP has at least two important technical advantages over FTP and FTPS.

SFTP is better with firewalls

FTPS can be painful to get working with firewalls. This is because directory listings and file transfers are made on a new network connection that is separate to the control channel on port 21. By default, firewalls will not permit these connections in FTPS (although it will usually work with FTP as firewalls are able to inspect the network traffic and open the appropriate port in advance). Instead, the firewall and the server must be configured for a certain range of ports for data transfer, which can get complicated.

By contrast, SFTP just works with firewalls. Data and commands are both sent over the standard port 22, which is usually enabled with firewalls by default. This is a significant advantage over FTP.

SFTP doesn’t use certificates

FTPS uses certificates to identify the server to the client. Server identification is important, as it is how the client verifies that it is connecting to the correct server. To be useful, however, certificates must be issued by a certificate authority – an organization that is authorized to issue them. Obtaining a certificate can be expensive and time-consuming.

SFTP doesn't use certificates – the server is identified by its public key (which is what a certificate contains, so they are both ultimately using the same mechanism). So as long as a client has the public key of the server on hand, they can confirm the server is the correct one. The server's public key (unlike a certificate) can be generated by the organization, and a certificate authority is not required. This significantly reduces the amount of administration necessary to get a server up and running.

There are some advantages in having a recognized organization such as a certificate authority to issue certificates, but much of the time it is not necessary, particularly for internal projects.

Is there any downside to using SFTP?

The absence of certificates might be an issue if you want the recognition of a certificate authority, but the main disadvantage of SFTP is that it is a complex protocol that is difficult to implement. Writing an SFTP client or an SFTP server is not an easy task.

This, however, is very unlikely to affect organizations using SFTP as part of their infrastructure – a large variety of clients and servers are available on various platforms, and they need only select the most suitable applications. All clients and server should interoperate, so there is considerable latitude in the choice of products. It is likely that features additional to the protocol will dictate the final selection.

Conclusion

This ebook has explained how SSL/TLS and SSH work to secure data being transferred over a network, and in particular the FTPS and SFTP protocols. While similar in functionality, FTPS and SFTP are vastly different in implementation. In a head-to-head comparison, SFTP comes out on top, although it may be wise to choose to support both protocols in your organization's technical infrastructure.

Complete FTP

Secure & reliable file transfer server for Windows

Thousands of companies worldwide rely on CompleteFTP to securely transfer their confidential files. It is packed with features that help you easily integrate secure file transfer into your business processes:

- easy to install and administer
- extensive range of features to suit small and big business alike
- highly customisable

> *We compared more than 10 products. CompleteFTP was by-far the winner on a cost/feature comparison.*
>
> **MSM Group** – Ohio, USA

Try it FREE for 30 days

completeftp.com