WS_FTP 7.6 Secret Key Exchange Brief

~~Confidential – Not for Distribution~~ (Expired)

Purpose

1. Briefly explain secret-key exchange and FTP Security protocols.

Terminology

- plaintext — Unencrypted data. The data does not actually have to be text. It may be an Excel document or an executable. It is called plaintext because it does not have to be decrypted before it is used. Actually, I don't know why they call it plaintext, that's just what they call it.
- ciphertext — Encrypted data. It may not have originated as text (see plaintext) but it is currently unintelligible.
- cipher — A cryptographic algorithm. (see next item)
- cryptographic algorithm. Also called cipher. A scheme of turning plaintext into ciphertext. An algorithm may be secret or it may be public.
- key — An additional piece of information that a cryptographic algorithm uses to transform plaintext to ciphertext and vice versa. Keys are usually very short compared to the data. Depending on the algorithm, the typical key is from 7 bytes (56 bits) to 128 bytes (1024) bits. Export restrictions at one point limited exportable algorithms to using 40 bit keys (5 bytes). The longer the key, the more difficult it is to crack the encryption.
- secret algorithm. In this case the algorithm used to transform the data and any necessary keys are kept secret.
- public algorithm. A well known algorithm that is used to transform data. All of the algorithms that we will be concerned with are public.
- secret key/symmetric key. The same key is used to encrypt and decrypt the data. This key must only be known to the two communicating parties.
- asymmetric keys. A key pair where the key used to encrypt the data is different from but related to the key used to decrypt. Special mathematical properties exist between the keys so that one may be used to encrypt and the other used to decrypt and get back the same result. One of these keys is usually made public while one is usually kept private.
- public key — The public key of the asymmetric key pair.
- private key — The private key of the asymmetric key pair.

Public vs. Secret Algorithms

Cryptographic algorithms take plaintext and turn it into ciphertext. They may use an additional piece of information such as a key. There are several methods of turning plaintext into ciphertext but most have their foundations in the two principles of substitution and transposition.

Imagine that I want to send you a message that other people will be able to see before it gets to you. We agree beforehand that I will move each letter forward by one place in the alphabet. z wraps around to a. My original message to you:

```
meet me at the red lion pub tonight
```

becomes

```
nffu nf bu uif sfe mjpo qvc upojhiu
```

unintelligible to the snoop. Obviously, we'll have to keep our little scheme to ourselves because if any one found out, they could decipher it just as easily as you could, and we might have some unexpected company. This is a secret algorithm. If the algorithm is known, the messages can more easily be broken.

Our little practice becomes common, and many people are now using this form of substitution cipher. So we decide to add a little twist. I now move each letter forward by an arbitrary number of places, but we keep the number secret. Shifting by 21 places (or backwards by 5), my next message to you:

```
last nite was heaven lets elope
```

becomes

```
gvno idoz rvn czvqzi gzon zgjkz
```

Now even though our snoop knows that we are using a simple shift substitution, he does not know how much to shift by. Our secret is safe*. This is an example of a public algorithm with a secret key. The algorithm is known, but the extra bit of information is kept secret.

*Unfortunately for the two star-crossed lovers in this example, they overestimated the strength of their cipher. It did not take the eavesdroppers long to try all 26 possibilities and discover the one that they used.

The advantage to using public algorithms is that they have been extensively analyzed by the best minds in the world and have been proven to be secure to the extent that you want to trust them. Millions of man and computer hours have been spent analyzing the ciphers and also the resulting ciphertext.

On the other hand, the security of a private algorithm is founded on the secrecy of the method. Because they are not subject to the same rigorous analyses, there may be weaknesses in the algorithm or patterns in the ciphertext that may allow the encryption to be broken. The usefulness of a private algorithm is also limited. Sharing any privately encrypted data with someone means sharing your algorithm with them. They are now equipped to break private data you share with anyone else (unless you have a different algorithm for each person).

Strong Keys

The key for the last message can be represented. (Letters in the top row map to the letter below it)

```
abcdefghijklmnopqrstuvwxyz
vwxyzabcdefghijklmnopqrstu
```

However, it may also just be represented as (+21). Each letter in the plaintext is the same distance from its corresponding letter in the ciphertext. Using our cipher, there are only 26 possible keys. The nominal strength of our cipher is just over 4 bits. The effective strength is actually less because of patterns in the data.

In general, we want it to take a long time for someone to try all the possible keys to decrypt our data. The more possible keys there are, the longer it will take to try them all. Cryptography expresses the strength of a cipher or key in bits. The strength is roughly the number of bits required to express the number of possible keys. For most ciphers, this is directly related to the key size. Patterns in the usable keys or in the ciphertext make the effective bit strength less than the key size so they are not exactly the same. Some fraction of the available keys for any algorithm should not be used because they result in data that is easily recognizable. In our case, we wouldn't want to use 0 (or 26) for our shift value, as it results in the same text.

A stronger substitution key for our jet setting partygoers would scramble the bottom array of letters as in the following example.

abcdefghijklm**n**opqrstuvwxyz
i**n**bmoh**a**rvxlzdfjkpcqugyswet

Now each letter is not shifted by the same amount. Here the a is shifted by +6 and the n is shifted by –12. This key may not be expressed by a single number from 1 to 26. It must be expressed by the entire arrangement of the bottom row (or its equivalent). There are 26! ways to arrange the bottom row of letters, which would give us 26! possible keys. For those of us who didn't sacrifice our personal lives to mathematics, 26! reads "twenty-six factorial" and is equal to the product of all integers from 26 to 1. I.e. 26 x 25 x 24 x 23 x 22 x … x 1.

26! = 403,291,461,126,605,635,584,000,000.

This is a much bigger number than 26. It has 27 decimal digits in it and would require about 54 bits to express. It would take a snoop a lot longer to try all of those possibilities.

Strong Ciphers

In reality though, a snoop would never try all those possible keys. The cipher itself exposes so many patterns in the ciphertext that he wouldn't have to. This one is ridiculously weak because we preserve the spaces. Other things it preserves are the relative distribution of letters in the English alphabet, the relative distribution of letters at the beginning or end of words, the relative distribution of two letter combinations, and the incidence of one and two letter words. This kind of attack is called "cryptanalysis", analyzing the ciphertext for patterns that can help with its decryption.

Real world ciphers must present no better way to decipher the data than to try all possible keys. All the ciphers used in the OpenSSL library have been analyzed, and to date, no serious weaknesses have been found that were not addressed in later versions of the algorithms. It is because of weaknesses (far more obtuse than this) that some of the algorithms have been deprecated and are no longer in widespread use.

Secret Keys

Imagine that:

- Two entities wish to communicate over the Internet via TCP/IP
- They agree on a protocol such as Telnet or FTP, and send plaintext data to each other according to the rules of that protocol. The protocol specifies things such as data formatting, maximum line length, what types of requests are allowed, and standard responses to those requests.
- Any other entity on the Internet that is between the communicating entities can see and capture all the traffic that passes between them. This will never change. However, when they are communicating using plaintext, these entities can also understand and manipulate the data, even so far as to change the data before sending it along.
- To secure the data, each party must encrypt their plaintext into ciphertext that may only be decrypted by the other party. To do this, they first agree on a cryptographic algorithm (cipher) and a key that they both will use. They now encrypt their data and send the ciphertext over the Internet. It may still be intercepted but it is now unintelligible to eavesdroppers. When each receiver gets the ciphertext, he uses the converse cryptographic function and the same secret key to turn the ciphertext back into plaintext.

The problem here of course is the exchange of the secret keys, and that is the subject of this brief. If you personally know everyone you will be communicating with, you can mail them the secret-key via U.S.P.S, or you can call them on the phone and dictate it to them. This becomes a problem when you communicate with many people, or if you don't know the people you'll be sharing data with.

The ideal is to find some way at the beginning of each conversation, to agree on an algorithm and a randomly chosen secret key to use for the rest of that conversation, bearing in mind that snoopers will be listening to the whole conversation, including the key agreement. That is where public key encryption comes in.

Public Keys

Imagine that:

- Two entities wish to communicate using FTP over the Internet, one is a server, and the other is a client.
- The server has an asymmetric key-pair, a public key and a private key. Now remember that if the public key is used to encrypt some data, it may only be decrypted by the private key.
- The client initiates the conversation by connecting to the server.
- The server sends the client the server's public key.
- The client chooses a secret-key algorithm and makes up a secret key. It combines the name of the algorithm and the key into a package we shall call a key-agreement.
- The client uses the RSA public-key algorithm to encrypt the key-agreement with the server's public key.
- The client sends the server the encrypted key-agreement.
- The server decrypts the key-agreement with its own private key.
- Tada! Now they both know what secret-key algorithm and key to use to continue their conversation.

Any eavesdropper that snooped the above conversation would capture the server's public key and the key-agreement that was encrypted with that public key. However since the eavesdropper does not have the server's private key, he cannot decrypt the key-agreement.

This is important. It does not matter that the eavesdropper has captured the public key. As a matter of fact, public keys are usually just that. Public. They are registered, published, distributed, whatever. If you ask for it, you can have it. But you cannot use it to decrypt what it was used to encrypt.

Bulk-Encryption

Now you may be wondering, why have secret keys at all? Why won't the following scenario work?

- Both entities have an asymmetric key pair.
- Whenever the client sends a message to the server, it encrypts it with the server's public key and sends it to the server. It may only be decrypted by the server with the server's private key.
- Whenever the server sends a message to the client, it encrypts it with the client's public key and sends it to the client. It may only be decrypted by the client with the client's private key.

Actually it would work. But it's just not practical. Asymmetric encryption takes hundreds of times longer than single key encryption. They also use much longer keys (at least 1024 bits) as they are the focus of more intense scrutiny. Public key encryption is best suited for special purposes such as key agreement and digital signatures.

The secret key algorithms used in the rest of the conversation are often called bulk-encryption ciphers, as they are the ciphers used to encrypt and decrypt the large amounts of data.

Certificates

The example above is a little simplistic about describing the key exchange. In actuality, it is more like a stork story compared to what actually goes on. The key exchange would follow a strict protocol such as TLS, SSL, SSH or IKE (Internet Key Exchange) and would involve the exchange of one or more certificates.

A certificate is a data file that contains information about somebody. I'll use the term somebody to avoid using the term entity. Somebody may be an individual, a business or an organization. Typically it has their name, maybe their address and contact info, but the two most important things in a certificate is the owner's public key and the digital signature of a trusted certifying authority.

In the above example, when the server sends the client its public key, what it is really sending is its certificate, which contains its public key. The client extracts the public key to perform the key exchange. The client will also check the information in the certificate to determine that it is talking to the correct server and not somebody else pretending to be that server.

You may now wonder what is to stop an impersonator from faking a certificate. That is, the impersonator takes the information from the server's certificate (which is publicly available), and replaces the public key with his own. The impersonator then intercepts a connection attempt from the client to the server and responds with his fake certificate. The client, duped into believing it is talking to the real server, performs the key agreement with the impersonator's public key and then sends the impersonator some confidential data.

This is made difficult by the presence of a certifying authority's digital signature. But before I can explain digital signatures, I must explain the secure hash function.

Secure Hash Functions

A hash function simply takes input data of any size and returns a digest of that data in a fixed size. Secure hash functions have special properties: (Programmers please don't confuse with container class hashing)

- It is repeatable. If you do it again with the same input, you will get the same output.
- The size of the digest is constant. The two most popular hash functions in use by SSL/TLS produce digests that are 16 bytes (for MD5) and 20 bytes (for SHA). MD<n> stands for Message Digest and n is the version number of the algorithm, currently at 5. SHA stands for Secure Hash Algorithm.
- It is one way. Not only is it virtually impossible to take a digest and recover the input data, but it is also virtually impossible to find any other message that hashes to that digest.
- It is collision free. It is virtually impossible to find another message that hashes to the same digest. Weakly collision free means it is virtually impossible to find a second message that hashes to the same digest as a particular message. Strongly collision free means it is virtually impossible to find any two messages that hash to the same digest. Both SHA and MD5 are strongly collision free.

Digital Signatures

A handwritten signature on a physical document usually means that the signer has read and either agrees to or verifies the contents. It is very difficult for the signer to deny having signed the document once subjected to handwriting analysis. These concepts carry over to digital documents.

Digital signatures use hash functions and public key encryption, but they use it in the reverse. Remember that data encrypted with the public key may only be decrypted with the private key. Well the reverse is also true, data encrypted with the private key may only be decrypted with the public key. Digitally signing a document involves the following steps.

- Compute a digest of the document using a hash function.
- Append the digest to the document.
- Encrypt the digest with your **private** key.
- Append the encrypted digest to the document.
- Append some information about who you are, what hash function you used, and what public key algorithm you used.

Congratulations, you just signed your first digital document.

Verifying the signature on a digital document is equally simple.

- Extract the encrypted digest and the unencrypted digest from the document.
- Compute a digest of the remainder of the document using the same hash function that the signer used.
- Decrypt the encrypted digest with the signer's **public** key and using the same public key algorithm.
- If all of the digests match, then the document was indeed signed by said signer (or someone in possession of his private key).

The digital signature on a certificate allows a client to check with a certifying authority, that this certificate did come from the expected server and not from an impersonator. After the bona-fide server has created his certificate describing himself, he sends it to a certifying authority to be signed. This authority signs it and sends it back.

When a client connects, the server sends his signed certificate. The client verifies the signature of the certifying authority, and if he trusts that authority, he goes ahead with the connection.

If an impersonator creates a fake certificate, he would have to somehow coax the certifying authority into signing it for him. This is generally regarded as difficult.

Certifying Authorities

The certifying authority in question must be someone that both the client and the server trust. For secure communications with the public, it must be someone that everybody trusts. Possibly the most trusted certifying authority today is Verisign. If an impersonator could take the public certificate of a store such as amazon.com, replace the public key with his own, and then convince a friend at Verisign to digitally sign it, he could impersonate amazon.com over the web, and possibly collect credit card numbers.

For this reason, secure communications all over the world no matter how algorithmically secure, all hinge on Verisign's secure business practices, and their protection of their private keys. To get a certificate of any importance signed by Verisign involves positive proof of identity.

Self Signed Certificates

Using freely available programs, it is easy to create a certificate authority, and sign your own or anybody else's certificates. In this case, it is still impossible for the impersonator to fake the certificate. Now instead of not being able to fake Verisign's signature, he cannot fake your CA's signature. Typically, your CA will have a different pair of keys than your server.

What if the impersonator created his own CA and signed his own fake certificate? This is where the issue of trust and certificate chains becomes important.

Certificate Chains

Ask any married couple. Trust is the most important thing. Without it, the relationship breaks down. This is the same situation facing digital communications. The client must trust the final certifying authority. Period. Or it closes the socket. The big question now is "Who do you trust?"

Well, for starters, you trust the owners of the digital certificates stored in your trusted certificates database. Most people trust Verisign because their software probably came with a Verisign public certificate installed in their database.

When your client software is presented with a certificate it does not trust, it may ask the user if it wants to trust the certificate. If the user says yes, the connection will proceed. The user may also have the option of adding that certificate to its trusted certificates database.

What if the client does not trust the issuing CA directly? Maybe the issuing CA itself has a certificate signed by a more well known and trusted CA. That is called a certificate chain. Security protocols provide for repeated certificate requests where the server may keep supplying certificates until the client sees one that it is happy with.

The important thing to remember is, you have to be careful who you let into your house, because they may let other people in. The certificates that you trust today influence the Internet sites you trust tomorrow.

Message Authentication

Now that you have seen the hash function used in digital certificates, you can see it used to solve a problem you may or may not have thought of. Message integrity. During secure communication (and especially during the key agreement phase) messages are subject to tampering by intermediate parties. For example, if a customer connected to amazon.com insecurely and placed an order, an eavesdropper could not only capture her credit card number, but could change the shipping address on the order before sending it on to Amazon. During a key agreement, this type of attack may be used to confuse both parties into agreeing on much weaker encryptions than they are capable of. Another kind of attack against a secure connection is a replay attack. If an eavesdropper captured a conversation, even if he cannot decrypt it, he could possibly replay parts of the stored conversation back against either party to have some ill effect.

For these reasons most security protocols implement some kind of scheme where each receiver can verify that each packet received is the same as what was sent. Message Authentication Codes (MACs) are calculated for and sent along with each packet.

Security Protocols

It seems we took a long time getting to the meat of the matter which is the FTP related security protocols, but the good news is that after laying all that groundwork I can almost just say "All of the above." Almost but not quite.

Just as there are protocols to define TCP and FTP communications, there are protocols to define secure communications. Even though two entities may have very secure public-key and bulk-encryption algorithms, it won't do either of them any good unless they agree on what to expect from each other and in what order. The SSH, TLS and SSL protocols define how connections are established, the data that is sent and the order in which it is sent so that secure communications may take place.

In particular, the protocols define:

- The sequencing of requests and responses
- The layout and format of the requests and responses

- The layout and format of the certificates.
- The default or startup encryption, MAC and compression modes
- The timing of mode changes
- The allowable algorithms, and the ways in which they may be combined.
- How to handle anomalies in the data stream such as packets that fail to authenticate.

The protocol documents themselves are very detailed and they do what I just said they do. Only with a lot more words and acronyms.

SSL stands for Secure Socket Layer. There have been three major versions of SSL, including SSLv2 and SSLv3. The original SSL has been deprecated and is no longer in widespread use. TLS stands for Transport Layer Security. TLS 1.0 is based on SSLv3 and is partially backward compatible with SSL implementations, masquerading as SSLv3.1. TLS 1.0 allows for a fallback to SSLv3 if both parties do not understand TLS.

The current definitive reference for the TLS 1.0 specification is "draft-ietf-tls-rfc2246-bis-00.txt" by Dierks & Rescorla. It is dated February 2002 and expires in August 2002. The TLS specification is still in draft form but this document also covers the differences between TLS 1.0 and SSLv3 so it is a good SSL reference.

The OpenSSL libraries by Eric A. Young and Tim J. Hudson provide a neatly packaged and reusable pair of DLLs that are free for commercial or non-commercial use. These are libeay32.DLL and ssleay32.DLL. The libeay32.DLL library lays the foundation and provides general purpose cryptographic functions. The ssleay32.DLL library implements both the TLS and SSL protocols.

SSH stands for Secure Shell. The acronym SSH may refer to either the protocol SSH, a specific program called ssh, or loosely to a suite of programs for doing SSH. The SSH protocols are described in a number of internet draft documents maintained by the IETF SECSH working group. These documents are dated March 2002, and will expire in September 2002. The protocols have already been through several revisions and are currently at SSH 2.0

There is an OpenSSH project for SSH which is maintained by the OpenBSD project, but it is not as neatly organized as the OpenSSL libraries. There is no equivalent DLL that we can drop in and use. The resources for implementing SSH solutions are as follows:

- The original developer of SSH, Tatu Ylonen, has formed a company called SSH Communications Inc. They sell SSH programs and toolkits.
- The OpenBSD project has implemented a completely free Unix product called OpenSSH. To keep it completely free and exportable, it is developed entirely outside of the U.S. and does not use any patented ciphers.
- Putty. The putty suite of programs (which is completely free) includes a secure ftp console mode client. The client code integrates both the SFTP protocol, and the SSH transport over which it rides.
- Libeay32.dll. The crypto library from the OpenSSL project is an excellent foundation for any implementation of secure transmission code. It is free for commercial use.

We hope this brief aids in your understanding of the FTP security protocols.

--The WS_FTP Dev Team.