Høgskolen i Gjøvik

Data Communication And Network Security

Group Project

SSL - Secure Sockets Layer

Victor Rudolfsson - 120912
Tommy B. Ingdal - 120913
Dag Jahren - 120923
Halvor Thoresen - 120915

May 14, 2013

Contents

1 Introduction

1.1 History

In 1994 Netscape company developed the SSL protocol. This was made with the intention to make secure transmissions with an encrypted data path between a server and a client. To begin with, SSL was created mainly for web browsers and communications with servers. [?]
Netscape continued working on the SSL technology, and in 1995 they released SSL version 2.0. The problem with this version was that it had a lot of vulnerabilities which could have been exploited. Some of the weaknesses were even released in an article [?]. For example, one of the released vulnerabilities meant that if your host had already been compromised, SSL wouldn't do much to protect you. After a not-so-successful release of SSL 2.0, they reworked the protocol and tried to solve previous vulnerabilities.
In 1996 Netscape released SSL version 3.0. SSL 2.0 had a vulnerability which enabled outsiders to change/modify data during transmission, but this was fixed in SSL 3.0. Version 2.0 had MACs which was encrypted at 40-bit, while the new v3.0 keys were encrypted at 128 bits. Because of the improved security they implemented for authentication keys, SSL 3.0 was much more secure against i.e. hacking attempts. [?, ?]

1.2 Introduction

SSL is a protocol containing "rules" used for communication between a client and server. SSL uses authenticated and encrypted communication to establish a secure connection. The protocol runs as a layer between the Transmission Control Protocol and Application layer. The main "job" for SSL is to provide encrypt data being transmitted and decrypt data being received, which only the applications using it should be able to do. [?] It also provides detection of whether or not data has been changed/modified during transmission. Both parties agree on which algorithms to use for exchanging keys, authenticating one another, encrypting/decrypting the data, as well as checking its integrity; to make sure that detection of a third party changing/modifying the data will be possible. Because of the encryption SSL provides by the use of strong algorithms, it allows for secure data communication, denying a potential third party to access the transmitted data. [?] SSL uses authentication to make sure you know who you are communicating with, by the use of certificates and public key encryption, as well as allowing messages to be signed. [?] For an SSL connection to be established, it's required to go through the "handshake", in which the key exchange occurs. During this procedure, a pre-secret key is created and then subsequently turned into a master key to allow for symmetric encryption of the information transmitted through the SSL connection. [?] In this report, we'll try to explain these procedures, and we'll go over some of the algorithms used in both the handshake, as well as the encryption that follows.

2 Short Description Of The Protocol

2.1 SSL Handshake

When you visit a website using the https protocol [1] *(technically not a protocol, since SSL/TLS is just layered on top of the HTTP protocol)*, the client and the webserver tries to establish a secure connection using SSL/TLS [2]. But before the secure connection is established, we have to perform a SSL handshake.
However, in order to complete a SSL handshake, the client and the server has to complete a number of steps, some of which are optional. And if for some reason a problem occurs in the negotiation, the connection is usually dropped.

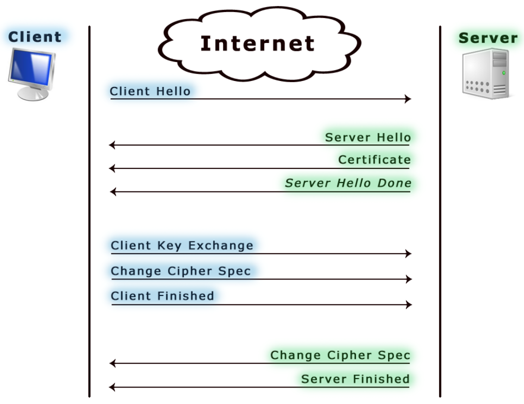


Figure 1: SSL Handshake

In this section we will give a brief overview of how the SSL handshake is completed and how it makes your communication with the web server secure.

2.1.1 9 Steps To Secure Communication

1. The client sends a ClientHello message to the server. This message contains information about which version the client supports, some randomly generated data and a list of all the cipher suites the client supports.

2. The server will now respond with a Server Hello message which include the highest version number both sides support, some randomly generated data and the cipher suite chosen by the server. It is worth mentioning that the cipher suite chosen by the server is the strongest suite both sides supports.

3. Now the server will send its certificate to the client. The server's public key is stored inside this certificate, and is used by the client to authenticate and encrypt the premaster key.

4. At this point the server is done, and waits for a response from the client.
5. It is now time to start exchanging keys. In step 1 and 2 both the client and the server sent some random values to each other. These random values is now used to generate the premaster secret. After the premaster secret is generated, it is encrypted using the server's public key and sent back to the server.
6. The client now sends a Change Cipher Spec message to the server, which basically says that all data being sent from the client from now on will be encrypted.
7. To finish the negotiation from the client's side, a Client Finished message is sent to the server.
8. The server now respond with a Change Cipher Spec message and tell the client that all data sent from the server from now on will be encrypted.
9. To complete the negotiation the server sends a Server Finished message back to client. And if everything went well the SSL handshake is now finished and a secure channel is initiated between the client and the server.

This is obviously a very brief overview of how the SSL handshake works. But in section 4, Bit-Level Description, we will go into more detail about the handshake, and also explain exactly what kind of data and values are being transmitted from the client to the server, and vice versa.

3 An Overview Of SSL

3.1 Global Description

3.1.1 Advantages

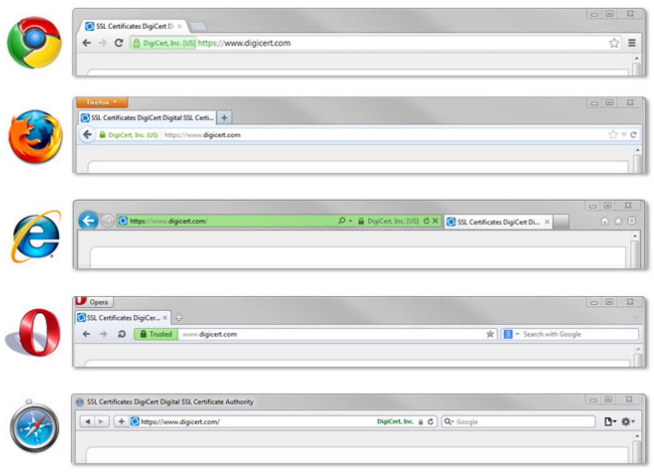


Figure 2: Visual cues [3]

The use of internet has drastically increased over the years and so has the different uses of the internet. This has demanded a huge increase in internet security. As a result of this the use of SSL and TLS has become very popular and quite normal. Even though these security protocols are used all over the internet, it's still a topic almost nobody got any knowledge of. So how does SSL help a website if the users don't know how it works, or what it does?
Well, the whole SSL certificate system is built around trust. The only thing the users/customers have to know, is that if a website is secures with SSL, it means that it is safe to use that website. Since the most important part for a website is to show the users that it is in fact safe to use, the universal symbol of safety, a lock, or the color green is added to the address bar of any website that has a valid SSL certificate, as seen in figure 3. This ensures the users that a certificate authority, or CA for short, has reviewed the website and made it safe to use and can be trusted. [4]
But why is it so important for a company's and websites to have the users trust them? The media has for the last couple of years focused a lot on internet security. Because of this more and more people are aware of the risk they take when sending information over the internet. And if they can ensure their users that it is in fact safe, then it is more likely that the user will use the services on their website.
Another important advantage for a company to have a SSL certificate is that it reduces the amount of successful phishing sites. This is because it is very difficult for criminals to get a valid SSL certificate on their phishing site. [5]

3.1.2 Disadvantages

Even though there are a lot of advantages with using SSL, there are also a couple of disadvantages that companies have to consider before getting a SSL certificate. One of the biggest disadvantages is the fact that getting a SSL certificate can be quite costly. This is because SSL providers have to ensure that their customer's website is secured and validate their identity. And since most companies buy SSL certificates to get more customers and earn more money, it might not be worth getting because of the implementation cost. [5]
Another disadvantage is the fact that encrypting the information takes more resources, and decreases the server performance. Even though this decrease in performance is minimal, it might cause trouble for sites with a lot of traffic. There is also special hardware that can minimize the performance decrease, but that might again not be very cost efficient for the company. [5]

3.2 Certificate Authority

3.2.1 Introduction

When we talk about a Certificate Authority we talk about a trusted third party that issues Digital Certificates. These certificates are used to digitally sign messages that identifies the sender.
We can compare this digital signature with a written signature. The purpose is to guarantee that the sender actually is who it claims to be.
But since signatures can be forged, digital signatures offers a number of different encryption algorithms to minimize this risk and offer the client and server a safe way to communicate across the Internet.

3.2.2 Issuing A Certificate

Certificate Authorities offers a few ways to verify the validity of an entity when they request a certificate. One way is to do a domain validation - this is the least secure way of verifying that the entity actually is who it claims to be.
The second method is to do an extensive validation of the entity, thus ensuring the users that the website really can be trusted.

Domain Validation
Like it says above this section, domain validation is the least secure way of validating an entity.
This is because the CA (Certicate Authority) only uses the domain to verify an entity. Whatever information they find doing a WHOIS is included in the certificate, and thus trusted by the CA.
This means that even though the certificate still uses 128-bit encryption, phishers can obtain a certificate and hide their identity.
And if you combine this with a MITM (man-in-the-middle) attack, an attacker can use DNS poisoning and use a domain validated certificate for your domain and redirect users to a fake site, thus enabling the attacker to obtain and collect sensitive user information.

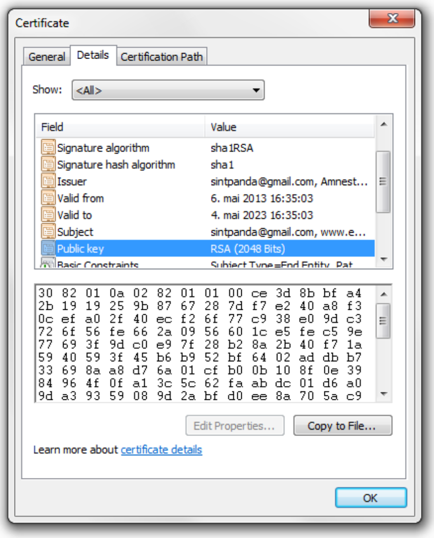
Extensive Validation
With an Extensive Validated Certificate you get alot more security for your users. EV Certificates are designed to prevent phishing attacks better than normal domain validated certificates.
The extensive validation [?] checks quite a few different aspects of your organization, including:

- Verifying that the organization is registered and active
- Verifying that the organization has exclusive rights to use the domain
- Verifying that the organization is not on any government blacklists

Even though this certificate is more expensive than the normal certificate, it is obvious that it provides many more benefits and that this certificate is the right choice if you really want to ensure the safety for your users.

3.2.3 Certificate Structure

In this section we will briefly go through the structure of an end user certificate and a root certificate. Even though the two certificates does have some fields which are the same, there are a few key differences between them.



Field	Value
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	sintpanda@gmail.com, Amnest...
Valid from	6. mai 2013 16:35:03
Valid to	4. mai 2023 16:35:03
Subject	sintpanda@gmail.com, www.e...
Public key	RSA (2048 Bits)
Basic Constraints	Subject Type=End Entity, Pat

```
30 82 01 0a 02 82 01 01 00 ce 3d 8b bf a4
2b 19 19 25 9b 87 67 28 7d f7 e2 40 a8 f3
0c ef a0 2f 40 ec f2 6f 77 c9 38 e0 9d c3
72 6f 56 fe 66 2a 09 56 60 1c e5 fe c5 9e
77 69 3f 9d c0 e9 7f 28 b2 8a 2b 40 f7 1a
59 40 59 3f 45 b6 b9 52 bf 64 02 ad db b7
33 69 8a a8 d7 6a 01 cf b0 0b 10 8f 0e 39
84 96 4f 0f a1 3c 5c 62 fa ab dc 01 d6 a0
9d a3 93 59 08 9d 2a bf d0 ee 8a 70 5a c9
```

Figure 3: Client Certificate

End user certificate
The end user certificate or, client certificate, basically is the certificate which gets installed on the client's computer.

Version

- This field contains information about the certificate's version. Today there are three versions; 1, 2 & 3. Higher certificate versions can contain more fields and information.

Serial number

- When the CA issues a certificate, they assign a unique integer to the certificate. This ID has to be unique for each certificate.

Signature algorithm

- Client and servers can communicate using a large range of different encryption algorithms. This field contains the algorithm identifier used by the CA to sign the content using the private key.

Signature hash algorithm

- This field contains information about which hash algorithm to use to hash the content before signing.

Issuer

- In this field we can see information about the entitiy that issued the certificate. For instance; "C = NO" means that the issuer is located in Norway (NO) and "O = Dundergruppen" is the name of the organization.

Valid from

- Information about the date the certificate is valid from.

Valid to

- Information about the date the certifiicate is valid to.

Subject

- This field contanins information about the CA that created, issued and signed the certificate.

Public key

- Here we can see the public key and information about the algorithm used.
  Example:
  RSA (2048 Bits)
  "*30 82 01 0a 02 82 01 01 00 ce 3d 8b bf a4 2b 19 19 25 9b 87 67 28 7d f7 e2 40 a8 f3 0c ef a0 2f 40 ec f2 6f 77 c9 38 e0 9d c3 72 6f 56 fe 66 2a 09 56 60 1c e5 fe c5 9e 77 69 3f 9d c0 e9 7f 28 b2 8a 2b 40 f7 1a 59 40 59 3f 45*

b6 b9 52 bf 64 02 ad db b7 33 69 8a a8 d7 6a 01 cf b0 0b 10 8f 0e 39 84 96 4f 0f a1 3c 5c 62 fa ab dc 01 d6 a0 9d a3 93 59 08 9d 2a bf d0 ee 8a 70 5a c9 3f 1f 15 b3 36 eb 67 15 e1 8e 5a c5 09 a7 84 bf 31 da a9 07 bf d4 e9 cd 2e 3f a5 e6 03 01 01 d6 67 b2 2b 92 84 06 84 f5 5f e0 d1 d7 5d e8 b0 e9 13 c5 07 db a3 86 5b 73 2f 07 f3 10 aa a4 79 ad e1 00 eb 8d 28 ce 3f ad f5 da 20 a6 f2 36 ef 88 b6 a7 93 9a f4 12 41 ca 99 96 5e 87 31 dd b5 78 d1 2d 83 bd 18 bc ac b2 f9 38 54 68 a3 b9 bb 3d ab 0a a3 6c e1 5c e7 7a d8 10 53 db bb 4d 1c 3d 2e 95 a2 0f 62 d6 ec 53 6f 38 25 02 03 01 00 01"

Basic Constraints

- If the certificate can be used as a CA it is specified in this field. If not it is specified that the certificate is an "End Entitiy".

Key Usage

- Contains information about which operations the public key can be used for. For instance: Digital Signature.

Thumbprint algorithm

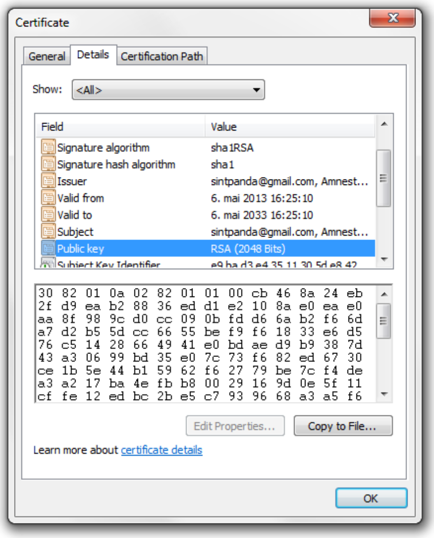
- Specifies the algorithm used to hash the certificate.
  For instance: SHA-1.

Thumbprint

- This is the hash itself. Basically used as an identifier.

Root Certificate

Like we mentioned earlier, end user certificates and root certificates contains many of the same fields and values.
It is however worth mentioning that the public key is exactly the same in both certificates. This because the client and server use a PKI (Public-key infrastructure)-modell.
However, there are differences, and in this



Field	Value
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	sintpanda@gmail.com, Amnest...
Valid from	6. mai 2013 16:25:10
Valid to	6. mai 2033 16:25:10
Subject	sintpanda@gmail.com, Amnest...
Public key	RSA (2048 Bits)
Subject Key Identifier	e9 ba d3 e4 35 11 30 5d e8 42

```
30 82 01 0a 02 82 01 01 00 cb 46 8a 24 eb
2f d9 ea b2 88 36 ed d1 e2 10 8a e0 ea e0
aa 8f 98 9c d0 cc 09 0b fd d6 6a b2 f6 6d
a7 d2 b5 5d cc 66 55 be f9 f6 18 33 e6 d5
76 c5 14 28 66 49 41 e0 bd ae d9 b9 38 7d
43 a3 06 99 bd 35 e0 7c 73 f6 82 ed 67 30
ce 1b 5e 44 b1 59 62 f6 27 79 be 7c f4 de
a3 a2 17 ba 4e fb b8 00 29 16 9d 0e 5f 11
cf fe 12 ed bc 2b e5 c7 93 96 68 a3 a5 f6
```

Figure 4: Root Certificate

section we will go through the field and values that differ between the two certificates.

Serial number

- This field contains a unique ID for this root certificate. This is used to distinguish between multiple certificates.

Subject Key Identifier

- This field contains an ID which provides a method of identifying certificates that contain a specific public key. A 160-bit SHA-1 hash of the value of the subjectPublicKey.

Authority Key Identifier

- To help verify the signature on this certificate, this field contains the public key used for that purpose.

Basic Constraints

- Since this is a Root Certificate this field contains the value "Subject Type=CA".

3.2.4 Providers

Worldwide there are a number of certificate authories, and the business is fragmented with national and regional providers.
However, certificates used for website security is largely held and issued by a small number of companies, and as of today more than 50 root certificates are trusted in the most popular web browers.

The market share between the top CA's, from W3Techs survey 2012, shows the following:

- Symantec (including VeriSign, Thawte & GeoTrust) with 42.9%
- Comodo with 26%

- GoDaddy with 14%
- GlobalSign with 7.7%

3.2.5 CA Compromise

A certificate authority compromise (CA Compromise) is an incident where an attacker is able to use the private key of a CA to issue fake certificates.
The worst case scenario is when a Root Certificate Authority is compromised. If that happen the Root Certificate Authority must notify all the relaying parties who trust their certificate, so that the relaying parties can stop trusting that specific certificate.

If however an Intermediate Root Certificate Authority is compromised, the Root Certificate Authority must revoke those certificates. And obviously the entitiy that got compromised must replace those certificates immediately, since those certificates no longer can be used due to the revocation.

DigiNotar Incident
On august 30, 2011, DigiNotar confirmed [6] that they had been compromised and that Google.com certificates was obtained by the attackers.

"*What at first appeared to be a one-off attack targeting Google Gmail users was actually part of a larger breach at Dutch digital certificate authority (CA) DigiNotar, which today confirmed speculation that it indeed was hacked and its SSL and EV-SSL CA system abused by attackers.*
"The company found out on July 19 that a hacking attempt had happened. At that moment, DigiNotar ordered an external security audit. This audit concluded that all fraudulently issued certificates were revoked. We found out yesterday, through Govcert, that the Google certificate was active. We revoked it immediately," said a spokesman today at Vasco Data Security International, of which the Dutch DigiNotar is a wholly owned subsidiary."

Consequences Of A Compromise
The CA paradigm is basically a chain-of-trust. So what happens when a CA experience a compromise?
If an attacker obtains a certificate for ex. Gmail.com, the attacker can basically impersonate Gmail by issuing forged certificates to its users with the purpose of stealing sensitive information and/or credentials - the trust is then broken.

Most operative systems include a feature that will automatically update the system when the developers issues a patch or an update. The common thing is to use certificates to ensure the integrity of the patch, so no malware get's installed on the users system.
What happens if an attacker can inpersonate Microsoft and digitally sign his own files using Microsoft's credentials? Well, if an attacker combines that with DNS Poisoning your operative system will see the file issued by the attacker as a legitimate file and install it, thus potensially installing malware on your system.
This can obviously be used to steal sensitive information, credentials and/or use the compromised computer in a botnet.

4 A Deeper Understanding

Since encryption of data is one of the responsibilities of the transport layer, it's important to mention that this is where encryption such as SSL/TLS (if used) operates. This has a very useful effect which benefits the protocol greatly: Because the transport layer is responsible for establishing connections as well, a connection will simply not be established if the security checks aren't passed.

The SSL / TLS procedure consists of two steps; the first one being authentication, and the second being encryption. Because SSL/TLS is based on the use of certificates, certificates is a very vital part of the process. One key difference (pun intended) between SSL and TLS, is that SSL requires the **server** to prove its identity to the client, whereas TLS requires both parties to prove their identity.

4.1 Handshake

The first thing that needs to be done to initiate encryption is what we call the 'handshake'. This is the first of the two processes that make up the SSL/TLS protocols, and in turn consists of several steps.

The first step in the handshake process is called the negotiation phase. During this phase, each packet contains a specific *message type code*, which indicates what type of packet is being sent.

First, the client sends a 'hello' to the server, this is called the *ClientHello* (message type 1). Listed within this 'hello' message is the algorithms (*Cipher Suites*) which the client supports for encryption, allowing the server to pick the strongest supported algorithm and reply with the server's equivalent to ClientHello, namely *ServerHello* (message type 2), containing a random number, the protocol the server has selected from the list of suggestions from the client, a random number, the CipherSuite, and compression method selected; followed by the server's x.509 certificate (message type 11) containing its public key. It's important to note that the server should always select the strongest available protocol.

When the negotiation is complete, the server will send a message indicating that it is done, called the *ServerHelloDone* (message type 14) packet, to which the client will respond with a packet called the *ClientKeyExchange* [12, 13] (message type 16). This packet contains either a *PreMasterSecret* [**?**] (encrypted with the public key found in the server's certificate), a public key, or nothing at all depending on the selected protocol [14, 15].

The random numbers contained within the *ClientHello* [16, 17] and *ServerHello* [18,19] will be used to calculate a *MasterSecret* together with the *PreMasterSecret* in the next step. The *MasterSecret* becomes a shared secret between the client and the server, which the data required for encryption will be based on.

When this is complete, both parties are ready to start encrypting data. This begins after the client sends a *ChangeCipherSpec* [20,21] packet, essentially indicating that *"Now this side is ready to change to the cipher suite we have agreed to use, with parameters we calculated",* and finally a *Finished* [22, 23] message containing the *message authentication code* [**?**] and hash calculated from the previous handshake messages.

Upon receiving this from the client, the server in turn responds with a similar *ChangeCipherSpec* message indicating that it, too, is ready to start encryption of data, followed by a *Finished* (message type 20) message.

Let's have a closer look at each of these messages:

ClientHello [16, 17] This packet is required to be sent by the client as soon as it connects to the server, but can also be sent as a response to a ClientHelloRequest packet (such a packet is sent by the server if it wants to initiate renegotiation of the session - it's the server's way of saying "Hey, can we renegotiate? You start!" - but it's the client's responsibility to start the process with a ClientHello message). This packet should contain the following values/parameters:

- cipher_suites: This parameter consists of the list of *cipher suites* supported by the client, ordered by the client's preference (first to last). If none of the cipher suites suggested by the client is supported by the server, a handshake failure alert will be returned and the connection will be closed. We'll elaborate further on what a cipher suite actually in following sections.

- client_version: This parameter indicates which the most recent version of the SSL/TLS protocol the client supports is (actually the version the client wishes to use, but these should be synonymous, as the client is required to suggest using the most recent version it supports).
- compression_methods: This parameter contains a list of compression methods supported by the client, ordered first-to-last by the client's preference.
- extensions: This parameter is optional, and may be sent if the client wishes to request extended functionality. Servers are required to accept ClientHello packets both *with* and *without* this field supplied, but it's up to the client to decide whether or not it wants to abort the connection if the server does not support the requested extended functionality. Each extension suggested by the client should contain two fields, these being:
  - extension_type: The type of the extension being suggested
  - extension_data: Specific information regarding the specified extension type.
- gmt_unix_time: This value contains the current UTC time stamp of the client's internal clock.
- random_bytes: This value should be 28 randomly generated bytes by a secure random number generator. This value together with the random_bytes value makes up the *random parameter*.
- session_id: This value holds the session ID, or nothing if it's either 1) not yet defined by the server (i.e. a handshake has not been completed) or 2) the client wants to renegotiate its security parameters.

ServerHello [18, 19] This packet is sent as a response to the ClientHello message, after the server has chosen which algorithms it wishes to use from the suggestions made by the client in the preceding ClientHello message. This packet, not unlike the ClientHello, carries some data with it - although rather than suggesting several alternative algorithms, it confirms which ones will be used. Let's have a closer look at these fields:

- cipher_suite: This field should contain the cipher suite selected by the server from the list supplied by the client in the preceding ClientHello message.
- compression_method: Similarly, this field should contain the compression method selected by the server from the list of suggestions sent with the ClientHello message.
- extensions: This field should contain a list of extensions supported by the server - if (and only if) they were suggested by the client in the ClientHello message. It should serve as a response to the client's way of saying "I request the following additional features: ", by saying "... and of those you requested, I support these: ".
- gmt_unix_time: The current UTC time stamp of the server's internal clock.
- random_bytes: 24 bytes randomly generated by a secure random number generator. Just like in the ClientHello message, this value together with the gmt_unix_time value makes up the *random* parameter.
- server_version: This field contains that suggested by the client in the preceding ClientHello message, and the highest version of the SSL/TLS protocol supported by the server.
- session_id: This field contains the unique ID for this session. If one that exists in the server's session cache was suggested by the client in the preceding ClientHello message and the server is willing to resume that session, it will respond with the same session ID. However, if that is not the case a new value will be created - unless the session cannot be resumed, in which case this field will remain empty.

ServerHelloDone [24, 25] This message is sent by the server when it's done sending the ServerHello and messages associated with it. It indicates that no more messages will be sent to support the key exchange, and the client is free to continue with its phase of the key exchange. When received by the client, it should verify that a valid certificate was provided if one was required, and that all other parameters in the ServerHello message were okay.

Client Key Exchange Message [12,13] If the client provides a certificate, this message will be sent right afterwards; otherwise this message will be sent as soon as the client receives the *ServerHelloDone* message.
This message sets the premaster secret by either sending the domain parameters (when Diffie-Hellman is used) or sending the secret encrypted with RSA. This message may also be sent without any data whatsoever, in the case the client sends a certificate containing a static DH exponent.

ChangeCipherSpec [20,21] This message is actually its own protocol, although it consists of only one single byte indicating that now is the time to start encrypting messages using 1) the information both parties have agreed on and 2) the required calculated information.

Finished [22,23] This message *always* follows a ChangeCipherSpec message, and is the first message encrypted using the agreed-on algorithms, keys and secrets. When received by a party, it must validate that the contents are correct before it's allowed to start sending and receiving data over the newly established secure connection. To be able to validate the contents of a Finished packet, we specification of its content must of course be provided:

- finished_label: This field contains either "client finished", or "server finished", depending on who sent the message.
- verify_data: This field should contain the output from a **pseudo-random function** that has been passed the *master secret*, *finished label* and a hash of **all** previous handshake message up to the Finished message.
  Note: The length of this message varies depending on the cipher suite after TLS version 1.1, but in previous versions this message had its size fixed to 12 bytes.

4.2 x.509 Certificate

The certificate [26] provided by the server is of the type x.509 as specified in RFC5280 (and earlier RFC2459) and updated in RFC6818. It is issued by a Certificate Authority, which [supposedly] proves the server's identity as verified by said authority, and contains a number of fields – in this case used to verify the legitimacy of the other party, as well as generating a private key.

x.509 certificates follow a very specific structure [27], expressed in ASN.1 (Abstract Syntax Notation One), the structure of which we described earlier in this document.

4.3 Cipher Suites

A *cipher suite* is a collection of algorithms which will be used for creating a pseudo-random number, establishing the keys used, encrypting the actual message, creating a hash digest of the message, as well as verifying these. The cipher suites used as part of the SSL (and/or TLS) protocol is defined by a cipher suite code. Initially, the cipher suite used is set to *TLS_NULL_WITH_NULL_NULL*. This must be changed for a secure connection to be established, as it provides no further security than an unsecured connection. As we will see, the format of these cipher suites is as follows:
Ex: [PROTOCOL TO USE]_[KEY EXCHANGE METHOD]_WITH_[METHOD OF SYMMETRIC ENCRYPTION]_[INTEGRITY CHECK]
For example, *TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA* would mean that we're using the TLS protocol with Diffie-Hellman RSA for key exchange, Triple-DES in EDE (*Encrypt-Decrypt-Encrypt*) mode and *Cipher Block Chaining*, and SHA to calculate the message digest used for the integrity check.

The following codes and their respective cipher suites are available in SSL, and we'll go through some (but not all) of these algorithms below [15, 28]:

For RSA:

Hex. Code	Cipher Suite
0x00, 0x01	TLS_RSA_WITH_NULL_MD5
0x00, 0x02	TLS_RSA_WITH_NULL_SHA
0x00, 0x3B	TLS_RSA_WITH_NULL_SHA256
0x00, 0x04	TLS_RSA_WITH_RC4_128_MD5
0x00, 0x05	TLS_RSA_WITH_RC4_128_SHA
0x00, 0x0A	TLS_RSA_WITH_3DES_EDE_CBC_SHA
0x00, 0x2F	TLS_RSA_WITH_AES_128_CBC_SHA
0x00, 0x35	TLS_RSA_WITH_AES_256_CBC_SHA
0x00, 0x3C	TLS_RSA_WITH_AES_128_CBC_SHA256
0x00, 0x3D	TLS_RSA_WITH_AES_256_CBC_SHA256

For Diffie-Hellman:

Hex. Code	Cipher Suite
0x00,0x0D	TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
0x00,0x10	TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
0x00,0x13	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
0x00,0x16	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
0x00,0x30	TLS_DH_DSS_WITH_AES_128_CBC_SHA
0x00,0x31	TLS_DH_RSA_WITH_AES_128_CBC_SHA
0x00,0x32	TLS_DHE_DSS_WITH_AES_128_CBC_SHA
0x00,0x33	TLS_DHE_RSA_WITH_AES_128_CBC_SHA
0x00,0x36	TLS_DH_DSS_WITH_AES_256_CBC_SHA
0x00,0x37	TLS_DH_RSA_WITH_AES_256_CBC_SHA
0x00,0x38	TLS_DHE_DSS_WITH_AES_256_CBC_SHA
0x00,0x39	TLS_DHE_RSA_WITH_AES_256_CBC_SHA
0x00,0x3E	TLS_DH_DSS_WITH_AES_128_CBC_SHA256
0x00,0x3F	TLS_DH_RSA_WITH_AES_128_CBC_SHA256
0x00,0x40	TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
0x00,0x67	TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
0x00,0x68	TLS_DH_DSS_WITH_AES_256_CBC_SHA256
0x00,0x69	TLS_DH_RSA_WITH_AES_256_CBC_SHA256
0x00,0x6A	TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
0x00,0x6B	TLS_DHE_RSA_WITH_AES_256_CBC_SHA256

Note: As we can see above, more recent versions of the SSL protocol use 3DES (or *Triple-DES*) rather than the traditional DES. Later on in this document, we'll thoroughly explain the DES algorithm, although not the 3DES algorithm which is the one actually used in more recent versions. However, we'll briefly touch the topic of additions and/or modifications done to the original by the more recent 3DES.

4.4 Authentication and Key Exchange

SSL relies mainly on two different algorithms for generating public and private keys, these being Diffie-Hellman, and RSA. However, it also uses some variations of these two algorithms - therefore, we'll go through the basics of both RSA and Diffie-Hellman below, and continue to mention the variations on each.

4.4.1 Diffie-Hellman

The Diffie-Hellman key exchange was published in 1976 in a ground breaking paper by Whitfield Diffie and Martin Hellman [29]. Essentially, this algorithm is together with the DES responsible for bringing cryptography to the public eye.

The algorithm devised is a way to generate an identical shared secret without any prior secrets. The algorithm's strength is based on the *discrete logarithm problem*, making it very easy to do one way, but very difficult to reverse. The process does, however, have some prerequisites; these being that the random number g must be lower than the prime number p, and the prime number p must be greater than 2.

The Diffie-Hellman key exchange is based on two parties each agreeing on a randomly generated integer, and a prime integer. Once both parties have agreed on these, they each generate a private integer (which must be less than the prime) which will become the private keys.

Each party proceeds to raise the generated number to the power of their private number, modulo the prime number they both agreed on, to create their *public key.* Thus, each party continues the procedure by sharing their newly created *public* *key* with one another, and use this to calculate their *shared secret.* This is done by raising the *other party's* public key to the power of their *private keys* modulo their agreed on prime. [30]

For example, let's assume *Alice* and *Bob* agree on using a prime, p, with the value 83, and a generated value g of 58.

Prime number (p):	83
Generated number (g):	58

Alice			**Bob**	
Private key (a):	30		Private key (b):	65
Public key *(g^a mod p)*	58^{30} mod 83 = 9		Public key *(g^b mod p)*	58^{65} mod 83 = 22
Bob's public key B	22	$\longleftrightarrow$	Alice's public key A	9
Shared key *(B^a mod p)*	22^{30} mod 83 = 29		Shared key *(A^b mod p)*	9^{65} mod 83 = 29
Shared secret: 29				

4.4.2 Variations on Diffie-Hellman

In SSL/TLS, we have a few different implementations of the Diffie-Hellman key exchange, denoted as *DHE*, or *DH*, followed by the method of authentication (such as RSA, or DSS) since none is provided by Diffie-Hellman by default. **DHE** stands for *Diffie-Hellman Ephemeral* - *ephemeral* stemming from the Greek *'ephemeros'* and being synonymous to *'momentary'* - which suggests that the parameters used in this method, unlike in the static variety of Diffie-Hellman [31], change with each session as to provide better forward-security (for example, should the server's private key somehow become publicly known, all previous sessions won't be suffering a breach of security as the private key is temporary). The exchange of parameters when using Ephemeral Diffie-Hellman occurs in the *server key exchange message* [32], as opposed to being included in the server's certificate, as is the case when Diffie-Hellman is used in static mode.

4.4.3 RSA

The RSA method is similar, albeit different, from the Diffie-Hellman method – it's similar in the sense that it's based on the exchange of public keys which are used for encryption, and using private keys for decryption. Just as with Diffie-Hellman described above, we once again use the *private key* of each party to generate the public key.

The RSA algorithm builds on the principles of modular arithmetics, and especially on two key mathematical algorithms: *Euler's Totient Function* (used to calculate phi (ϕ) of a given number), and the *Euclidian algorithm* (to calculate the GCD of two given numbers). Let's see how these are used to create the key pair used for encryption the RSA way! [33]

$\phi(i)$ of a given integer i is the amount of numbers between 1 and $i - 1$ that are relatively prime – as in, the amount of number which has 1 as its only common divisor. Furthermore, $\phi(i)$ can be easily calculated if the given integer i is a product of two *different* primes, as such:

$\phi(P_1 * P_2) = P_1 - 1 * P_2 - 1$

For example, given the integer 39, we could calculate $\phi(39)$ like so:

$\phi(39) = \phi(3 * 13) = 3 - 1 * 13 - 1 = 2 * 12 = 24$

Now, *Euler's Totient Theorem* states that if given two numbers that are relatively prime, then when you raise the lower number to phi of the higher and divide the result by the higher number, you should always get the remainder 1. In mathematical terms, given the numbers N and M, where $N < M$ and $\gcd(N, M) = 1$, then $N^{\phi(M)} = 1 \pmod{M}$.

Now, we can also see that $N^{\phi(M)} * N^{\phi(M)} = 1 * 1 \pmod{M}$

Which is the same expression as $N^{2\phi(M)} = 1 \pmod{M}$

... and so on. Because of this, we can continue to increase the factor of $\phi(M)$, and still get $1 \pmod{M}$ as the result. This means that we can generalize this equation a bit to become:

$N^{k\phi(M)} = 1 \pmod{M}$

This would mean that N raised to any number k which $\pmod{\phi(M)}$ would become 0 equals $1 \pmod{M}$.

Therefore, if $\gcd(N, M) = 1, N < M$ then $N^S = 1 \pmod{M}$ when $S = 0 \pmod{\phi(M)}$

If we continue to multiply both sides of this equation with N, we see that $N^S * N = 1 * N \pmod{M}$ which is the same as $N^{S+1} = N \pmod{M}$

Now, if we instead find two numbers which we could multiply together to create $S + 1$, we could raise N to the product of those numbers to get N back again; but reversing that calculation to find what those two numbers were would be a lot harder.

Consider we find P and Q where $P * Q = 1 \pmod{\phi(M)}$, we see that $N^{(P*Q)} = N \pmod{M}$ (remember, since $P * Q = 1 \pmod{\phi(M)}$, raising N to the power of $P * Q$ is the same as raising N to the power of $S + 1$ as we mentioned above) and that $N^{(P*Q)} = (N^P)^Q$.

Now we have two operations here, first we have: N^P Then further raising that to the power of Q will give us $N \pmod{M}$ back again, but lets stop for a bit there - let's say that the cipher text c can be given from that first step; that would give us $N^P = c \pmod{M}$

Which we could then turn back into $N \pmod{M}$ like so: $c^Q = N \pmod{M}$

This is the foundation of RSA. P becomes our public key together with M, and Q becomes our private key together with M. [33, 34]
In the equation above, we can see that raising N to the public key produces the ciper text $c \pmod{M}$, and to get N back we raise the cipher text c to our private key Q and get $N \pmod{M}$.

Let's see how this would work in a more practical sense:
Let's say Alice wants Bob to be able to send her encrypted messages which only Alice can decrypt. Just like we previously mentioned in the Diffie-Hellman explanation, Alice needs to calculate a public key based on her private key. But unlike in the Diffie-Hellman example, Alice does not select a completely random number. Instead, she must find *two primes* to multiply together which will become M -

since we used the number 39 to explain ϕ earlier, let's work with that number again.

1. Alice selects two prime numbers, 13 and 3, to create M:

   $M = 3 * 13 = 39$

2. Alice then calculates $\phi(M)$ like so:

   $\phi(M) = 3 - 1 * 13 - 1 = 24$

3. Alice must then find P and Q where $P * Q = 1 \pmod{\phi(M)}$ or:

   $P * Q = 1 \pmod{24}$
   P and Q must be relatively prime to $\phi(M)$ (24), which means that $\gcd(P, \phi(M)) = 1$ and $\gcd(Q, \phi(N)) = 1$. Preferably, P and Q should also be relatively prime to one another.

4. Alice therefore lets her private key $Q = 53$ and now has the equation:

   $P * 53 = 1 \pmod{\phi(M)}$, or $P * 53 = 1 \pmod{24}$ which can be expressed as:

   $P * 53 = k * 24 + 1$ where k can be any number.

5. Alice sees that $Q = 5$ is one of the possible solutions to this, because $5 * 53 = 265 = 1 \pmod{24}$. The criteria $\gcd(P, Q) = 1$ is satisfied as $\gcd(5, 53) = 1$, and the greatest common divisor of Q and $\phi(M)$ is 1, and so is the greatest common divisor of P and $\phi(M)$, making them relatively prime.

6. Alice has now calculated her public key P to be 5, and her private key Q to be 53.

7. Alice sends Bob her public key, which Bob can use to encrypt messages.

Once again we see that the public key cannot be used to *decrypt* messages it has encrypted. However, Alice's *private key Q* can decrypt messages encrypted with her public key.

In a similar fashion, Bob would calculate his own public key and send to Alice, which Alice would use to encrypt *her* messages to Bob, as is the basis for all public-key encryption algorithms.

4.4.4 The Elliptic Curve variety

Whilst Diffie-Hellman and/or RSA as mentioned above are good, they're also rather old and people are familiar with how they work. In recent years, another field of cryptography has grown increasingly popular - namely Elliptic Curve Cryptography [38]. Whereas in the regular version of Diffie-Hellman we have the domain parameters $p > 2$ and $g < p$, the prime number and the generated number, in the Elliptic Curve version of i.e Diffie-Hellman, things work slightly different. [39] Instead, we have the domain parameters E which is the elliptic curve to use (for example p-256r1) [40] and G which is a base point on this curve. The curves used for this all have a set of predefined parameters, and the curves themselves are predefined by the Standards for Efficient Cryptography Group, or SECG. Whereas we won't go deep into the subject of Elliptic Curves in this report, we felt the need to mention their existence and purpose. For EC-Diffie-Hellman specifically, the difference from regular Diffie-Hellman is that knowing a curve E and it's parameters, as well as a base point G on this curve, the private key Q becomes a randomly chosen number between 1 and $N - 1$, where N is the order of G; and the public key P becomes the product of $Q * G$. It's also important to note that EC is not an encryption algorithm *in itself*, but rather a more recent way of exchanging and calculating necessary parameters which can be used together with more established encryption/key-exchange/authentication algorithms such as RSA, Diffie-Hellman, DSA or even combinations of these [41, 42]. One great benefit of using Elliptic Curves is the amount of data that needs to be communicated and calculated; which makes it especially useful for mobile devices where computational power is limited. For example, using Elliptic Curves we can get the same level of security as RSA with a 1024 bit long key, by using only a 160 bit sized key. [43]

4.4.5 Secrets

When the handshaking process has been performed (i.e using RSA or Diffie-Hellman), both parties become ready to create the *master secret.* This is what the *pre-master secret* is used for, and this pre-master secret is created differently depending on which key-exchange algorithm was used. In the case of Diffie-Hellman, the shared secret **becomes** the pre-master secret [35] as one might have already guessed, but what if RSA was used? As we saw above, RSA does not create a shared secret, but generates public and private keys. Therefore, in the case RSA was used it's the *client's* responsibility to generate a 48-byte pre-master-secret, then proceed to encrypt this using the server's public key and transmit it to the server [36]. Once received by the server, the server can decrypt it using its private key; and from here on, the process for generating the master secret is the same.

Now that both parties know the pre-master-secret, they can create the master secret by concatenating three MD5 hashes. These MD5 hashes are created in essentially the same way [37]:
$MD5(pre\_master\_secret{+}SHA(S{+}pre\_master\_secret{+}Random_{client}{+}Random_{server}))$

Where S is the letter 'A' for the first hash, 'BB' for the second hash, and 'CCC' for the third hash; $Random_{client}$ is the random parameter passed by the client in the *ClientHello* message, and $Random_{server}$ is the random parameter passed by the server in the *ServerHello* message.

Once these three MD5 hashes have been created and concatenated into one long string, both the server and the client share an identical *master secret* [37].

4.5 Encryption

DES, or the Data Encryption Standard, is an encryption algorithm using block cipher. It takes plain-text of 64-bits and encrypts it to a cipher-text of the same size. DES operates with a key, so the encryption/decryption will only be doable for those who know the key. This is a 64-bit key. [**?**]

In this example we use**'M'** as plain textmessage, and **'K'** to be the input for encryption/decryption key.

M = ABCDEFGH
M(hex) = 4142434445464748
M(binary)= 01000001 01000010 01000011 01000100 01000101 01000110 01000111 01001000

K = 0202020202020202 (in hex, used this number for simplicity)

K(binary)= 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

Creating 16 subkeys: The reason for making these sub-keys, is that they will be used later for the Feistel function. You have a input key **'K'** which give us a 64-bit binary key. To create the 16 sub-keys you first have to make a permuted key **'K+'**. This key will be permuted using a table **'PC-1'**. In the **'PC-1'** table the first entry is "57" [**?**]. This means the that first bit in the permuted key **'K+'** will become the 57th bit in the original key **'K'**. Because of the permutation, the new key will now be 56-bits. [**?**, **?**]

eks:
original 64-bit key:
K = 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

permuted 56-bit key:
K+ = 0000000 0000000 0000000 0000000 1111111 0000000 0000000 0000000

When the permuted key **'K+'** is made, it will be split into a left and a right half (C_0 and D_0). They will each have 28 bits now. [**?**]

C_0 = 0000000 0000000 0000000 0000000
D_0 = 1111111 0000000 0000000 0000000

Now it creates 16 blocks of **C** and **D** by rotating the bits to left, by a specified amount declared in the algorithm(see rotations in the key schedule [**?**]).

C_0 = 0000000000000000000000000000
D_0 = 1111111000000000000000000000

C_1 = 0000000000000000000000000000
D_1 = 1111110000000000000000000001

C_2 = 0000000000000000000000000000
D_2 = 1111100000000000000000000011

$C_3 = 0000000000000000000000000000$
$D_3 = 1110000000000000000000001111$

$C_4 = 0000000000000000000000000000$
$D_4 = 1000000000000000000000111111$

$C_5 = 0000000000000000000000000000$
$D_5 = 0000000000000000000011111110$

$C_6 = 0000000000000000000000000000$
$D_6 = 0000000000000000001111111000$

$C_7 = 0000000000000000000000000000$
$D_7 = 0000000000000000111111100000$

$C_8 = 0000000000000000000000000000$
$D_8 = 0000000000000011111110000000$

$C_9 = 0000000000000000000000000000$
$D_9 = 0000000000000111111100000000$

$C_{10} = 0000000000000000000000000000$
$D_{10} = 0000000000011111110000000000$

$C_{11} = 0000000000000000000000000000$
$D_{11} = 0000000001111111000000000000$

$C_{12} = 0000000000000000000000000000$
$D_{12} = 0000000111111100000000000000$

$C_{13} = 0000000000000000000000000000$
$D_{13} = 0000011111110000000000000000$

$C_{14} = 0000000000000000000000000000$
$D_{14} = 0001111111000000000000000000$

$C_{15} = 0000000000000000000000000000$
$D_{15} = 0111111100000000000000000000$

C_{16} = 0000000000000000000000000000
D_{16} = 1111111000000000000000000000

Now its time to form the actuall subkeys. The subkeys are formed by doing a permutation on each block pair C_nD_n.These keys are now 56-bit, so now it uses the **'PC-2'** [?] table. By using permutation on the block pairs it will make the keys 48-bit. [?, ?]

K_1 = 000000 000000 000000 000000 001000 100010 000110 000011
K_2 = 000000 000000 000000 000000 001001 100010 000100 000011
K_3 = 000000 000000 000000 000000 001001 100000 000101 000010
K_4 = 000000 000000 000000 000000 010001 001000 000101 000010
K_5 = 000000 000000 000000 000000 000001 001000 010001 001000
K_6 = 000000 000000 000000 000000 010010 001001 010001 001000
K_7 = 000000 000000 000000 000000 000010 001101 010000 101000
K_8 = 000000 000000 000000 000000 000010 000101 110000 100000
K_9 = 000000 000000 000000 000000 000010 000101 100000 110000
K_{10} = 000000 000000 000000 000000 100000 010100 100000 110000
K_{11} = 000000 000000 000000 000000 100000 010000 101000 010000
K_{12} = 000000 000000 000000 000000 100100 010000 001000 010100
K_{13} = 000000 000000 000000 000000 000100 010000 001010 000100
K_{14} = 000000 000000 000000 000000 000100 000010 000010 000101
K_{15} = 000000 000000 000000 000000 001000 100010 000010 000101
K_{16} = 000000 000000 000000 000000 001000 100010 000010 000011

Encryption of message M: First there is a permutation of M using the initial permutation [?] table. This will be done just like the subkeys was made, where it use the entries in table to get the correct bit. [?]

Eks:
M = 01000001 01000010 01000011 01000100 01000101 01000110 01000111 01001000

after doing permutation on **'M'**:
IP = 11111111 00000000 01111000 01010101 00000000 00000000 10000000 01100110

Then **IP** is devided into a left half and a right half, both value of 32 bits

L_0 = 11111111 00000000 01111000 01010101
R_0 = 00000000 00000000 10000000 01100110

what happens next is that the blocks created from **IP** will go through 16 iterations of Feistel and XOR operation. In the first round of iteration one of the blocks (**L** or **R**) will be sent as input in the feistel function together with subkey K_n, where **'n'** represent the round of iteration the process is at. The output of the Feistel function will then be XOR'ed with the other half-block (**L** or **R**, not the one that is used in Feistel). After this is done, the iteration goes to the next round. Now the half blocks switch place. So for the next round the results from the half-block XOR'ed with the output of Feistel, will be sent as input for Feistel. While the half-block from previous round who was sent as input for Feistel, will be XOR'ed with output of the new Feistel operation. [?, ?, ?, ?, ?]

Example:
Round 1:
R_0 is sent as input for Feistel, f (R_0, K_1)
then $R_1 = L_0 \oplus$ output of Feistel

Round 2:
R_1 is sent as input for Feistel, f (R_1, K_2)
$R_2 = R_0 \oplus$ output of Feistel

this repeats for 16 rounds. After the 16th round the two half-block will be combined again and will be going through a final permutation (**FP**) using the **FP** table [?]. Now the key is encrypted. To decrypt, it will go through the same iterations, with only minor changes. "The Feistel structure ensures that decryption and encryption are very similar processes — the only difference is that the subkeys are applied in the reverse order when decrypting." [?]

Feistel function: What happens in the Feistel function is that it first takes in a half block and a subkey as input. The half block is then going through Expansion. [?, ?] It is expanded from 32-bit to 48-bits. This is done by using

expansion permutation [?]. The output from expansion is then XOR'ed with the 48-bit subkey. After the XOR is done, the combined key will now be divided in 8 6-bit pieces. This will be the input for the Substitution boxes. "Each of the eight S-boxes replaces its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table." [?]. There is a substitution box for each of the 8 6-bit pieces [?]. By looking at the table you have a row which represents the outer bits and the column for the inner 4 bits of the piece. Example: 101010, 1 and 0 is the outer bits, and 0101 is the inner bits.
When the substitution is done on each of the 8 6-bits pieces, It will then use Permutation(**P**, [?]) on the 32 bits. The output from **P** will be the output from the Feistel function.

4.5.1 Variations on DES - Triple-DES

As computational power per dollar increases continuously, the protection provided by DES became less secure as time went by. To combat this, 3DES (or Triple-DES) was implemented and is just what it sounds like - DES, times three. Instead of 16 rounds, triple DES uses 48 rounds and a key size of either 56, 112 or 168 bits.

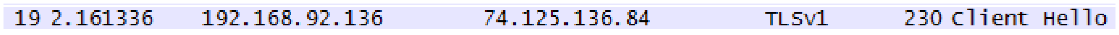
4.6 Integrity Check

SSL provides integrity verification by signing messages with a hash digest using either SHA or MD5 depending on the cipher suite in use. Because hash digests are essentially one way functions, and contain no random elements, it is possible to verify that data has not been tampered with by creating a hash of all other data contained within the message and comparing this against the hash in the message. If the data has been tampered with, the hash would come out as different, and it would be evident that something is wrong with the message.

However, this does not prevent somebody from tampering with the data and then updating the hash in the message; but it does provide some protection against casual or accidental tampering of the data.

5 Analysis Of The Recording

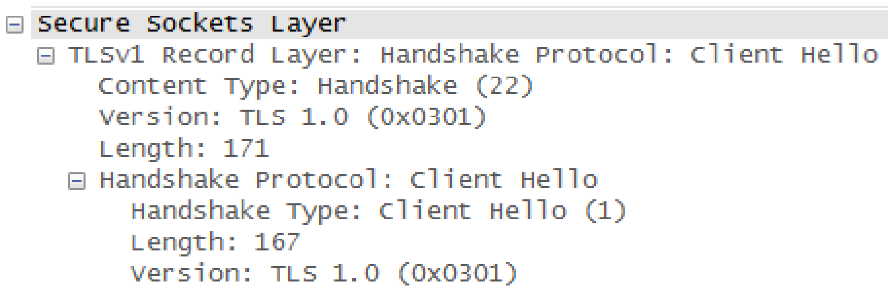
All the important parts of the recording:



```
19 2.161336    192.168.92.136        74.125.136.84         TLSv1       230 Client Hello
```

Figure 5: Client Hello

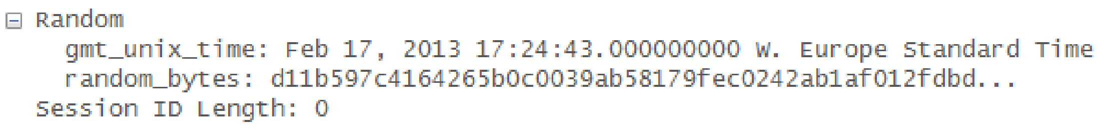
This is the Client Hello packet seen from wireshark. From this we can see the client's ip adress (192.168.92.136) and the server's ip adress (74.125.136.84) and that the recording used TLS instead of SSL.



```
⊟ Secure Sockets Layer
  ⊟ TLSv1 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 171
    ⊟ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 167
        Version: TLS 1.0 (0x0301)
```

Figure 6: TLS verson

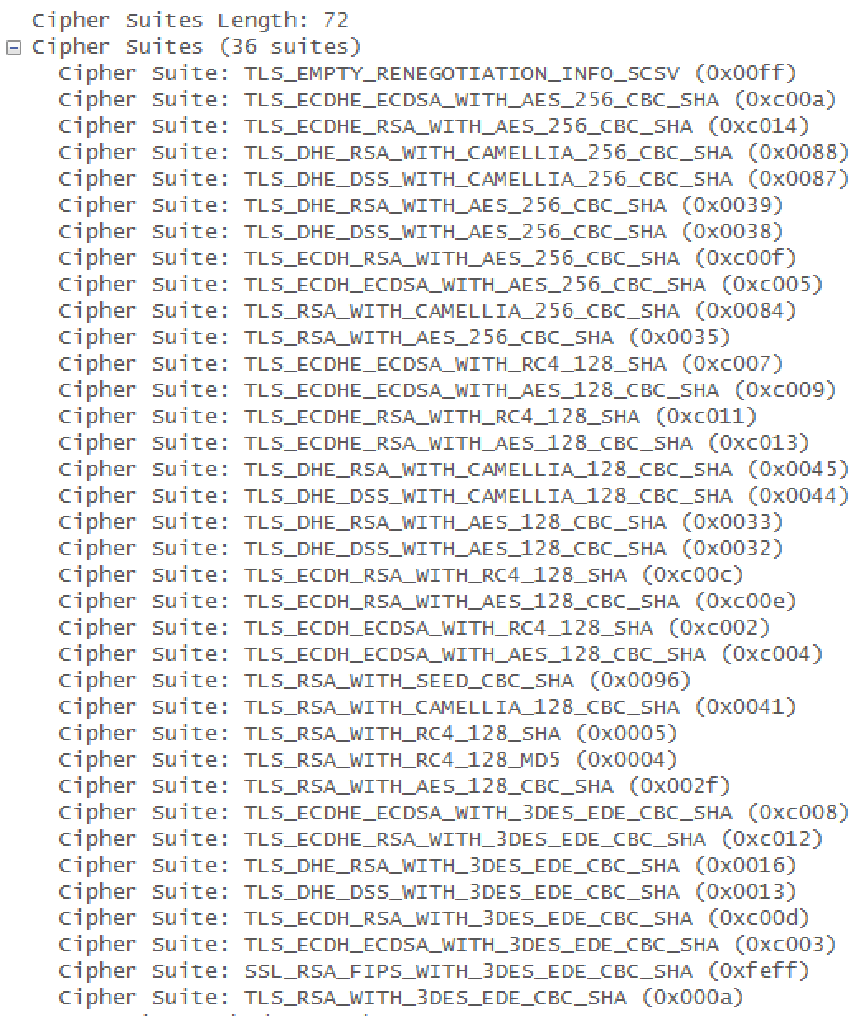
Version Number: Client sends the version number corresponding to the highest version it supports. In this case TLS version 1.0.



```
⊟ Random
    gmt_unix_time: Feb 17, 2013 17:24:43.000000000 W. Europe Standard Time
    random_bytes: d11b597c4164265b0c0039ab58179fec0242ab1af012fdbd...
  Session ID Length: 0
```

Figure 7: Client Random

here the cient generates a 32 byte value consisting of a 4-byte number that contains the client's date and time, plus a 28 byte randomly generated number that ultimately will be used with the server random value to generate a master secret. In this case we can see that the recording was done on feb 17 2013.

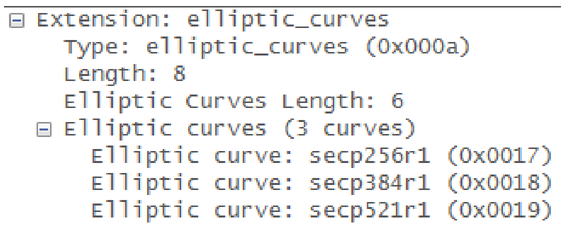


```
  Cipher Suites Length: 72
⊟ Cipher Suites (36 suites)
    Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0088)
    Cipher Suite: TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA (0x0087)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
    Cipher Suite: TLS_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0084)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
    Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x0045)
    Cipher Suite: TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA (0x0044)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
    Cipher Suite: TLS_ECDH_RSA_WITH_RC4_128_SHA (0xc00c)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_RC4_128_SHA (0xc002)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
    Cipher Suite: TLS_RSA_WITH_SEED_CBC_SHA (0x0096)
    Cipher Suite: TLS_RSA_WITH_CAMELLIA_128_CBC_SHA (0x0041)
    Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
    Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)
    Cipher Suite: TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012)
    Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
    Cipher Suite: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)
    Cipher Suite: TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA (0xc00d)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc003)
    Cipher Suite: SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA (0xfeff)
    Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
```

Figure 8: Client Cipher Suits

In this tab we can see that the client sends all the different cipher suits that it is compatible with. Every part of the name describes the different parts the suit is using. Taking TLS_RSA_WITH_DES_CBC_SHA as an example: TLS is the protocol version, RSA is the algorithm that will be used for the key exchange, DES_CBC is the encryption algorithm (using a 56-bit key in CBC mode), and

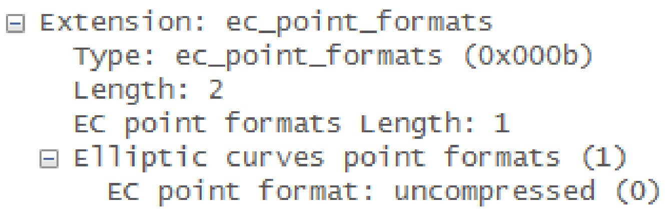
SHA is the hash function.



```
⊟ Extension: elliptic_curves
    Type: elliptic_curves (0x000a)
    Length: 8
    Elliptic Curves Length: 6
  ⊟ Elliptic curves (3 curves)
      Elliptic curve: secp256r1 (0x0017)
      Elliptic curve: secp384r1 (0x0018)
      Elliptic curve: secp521r1 (0x0019)
```

Figure 9: Client Curves

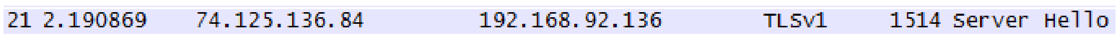
In this recording we can also see that elliptic curves are in use, and here the client sends three different curves that the client supports.



```
⊟ Extension: ec_point_formats
    Type: ec_point_formats (0x000b)
    Length: 2
    EC point formats Length: 1
  ⊟ Elliptic curves point formats (1)
      EC point format: uncompressed (0)
```

Figure 10: Client EC Point Format

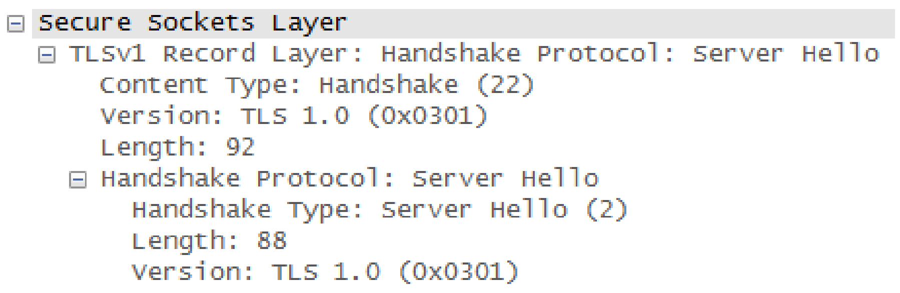
The client EC Point Formats is a list of the point formats a client is able to parse, which the client is required to send with the ClientHello message when it suggests the use of elliptic curves as is the case when using the Elliptic Curve variety of an algorithm such as Diffie-Hellman.



```
21 2.190869    74.125.136.84        192.168.92.136        TLSv1     1514 Server Hello
```

Figure 11: Server Hello

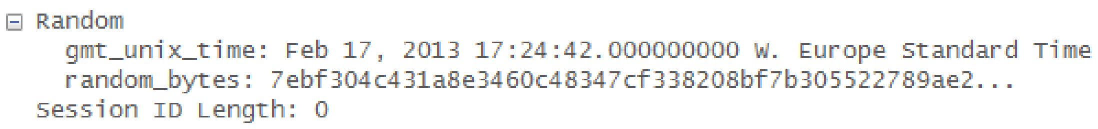
This is the server hello packet seen from wireshark.



```
⊟ Secure Sockets Layer
  ⊟ TLSv1 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 92
    ⊟ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 88
        Version: TLS 1.0 (0x0301)
```

Figure 12: Server TLS Verson

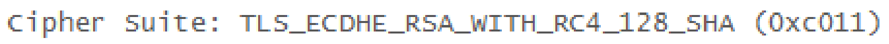
The server sends the highest version number supported by BOTH sides.



```
⊟ Random
    gmt_unix_time: Feb 17, 2013 17:24:42.000000000 W. Europe Standard Time
    random_bytes: 7ebf304c431a8e3460c48347cf338208bf7b305522789ae2...
  Session ID Length: 0
```

Figure 13: Server Random

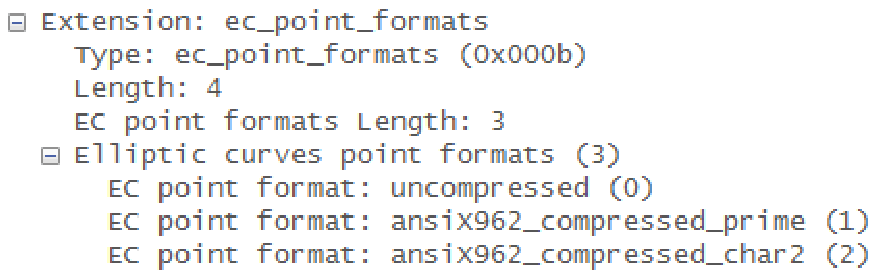
Here the server creates a random number the same way the client did earlier.



```
Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
```

Figure 14: Server Cipher Suite

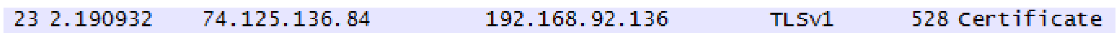
The server will choose the strongest cipher that BOTH the client and server support. In this case its TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011). If no cipher is supported by both the client and server, the session is ended with a "Handshake Failure" alert.



```
⊟ Extension: ec_point_formats
    Type: ec_point_formats (0x000b)
    Length: 4
    EC point formats Length: 3
  ⊟ Elliptic curves point formats (3)
      EC point format: uncompressed (0)
      EC point format: ansiX962_compressed_prime (1)
      EC point format: ansiX962_compressed_char2 (2)
```

Figure 15: Server EC Point Formats

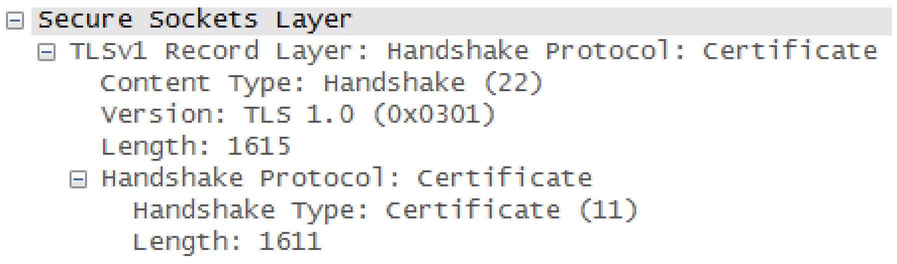
When the server has received a ClientHello containing a list of elliptic curves and chosen a curve it wishes to use, it's required to use the point format suggested by the client to describe the point on the curve it wishes to use, as well as the curve it wishes to use.



```
23 2.190932    74.125.136.84          192.168.92.136         TLSv1       528 Certificate
```

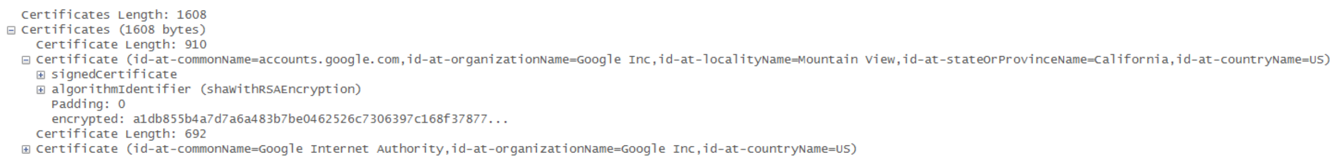
Figure 16: Certificate

The server now sends it's Certificate to the client.



```
⊟ Secure Sockets Layer
  ⊟ TLSv1 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 1615
    ⊟ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 1611
```

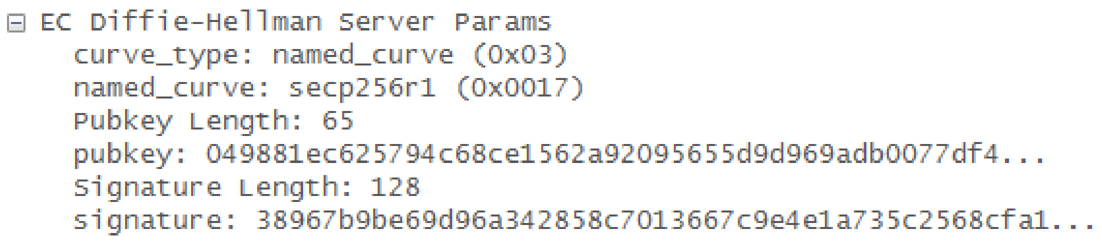
Figure 17: Certificate TLS verson



```
  Certificates Length: 1608
⊟ Certificates (1608 bytes)
    Certificate Length: 910
  ⊟ Certificate (id-at-commonName=accounts.google.com,id-at-organizationName=Google Inc,id-at-localityName=Mountain View,id-at-stateOrProvinceName=California,id-at-countryName=US)
    ⊞ signedCertificate
    ⊞ algorithmIdentifier (shaWithRSAEncryption)
      Padding: 0
      encrypted: a1db855b4a7d7a6a483b7be0462526c7306397c168f37877...
    Certificate Length: 692
  ⊞ Certificate (id-at-commonName=Google Internet Authority,id-at-organizationName=Google Inc,id-at-countryName=US)
```

Figure 18: Certificate Certificates

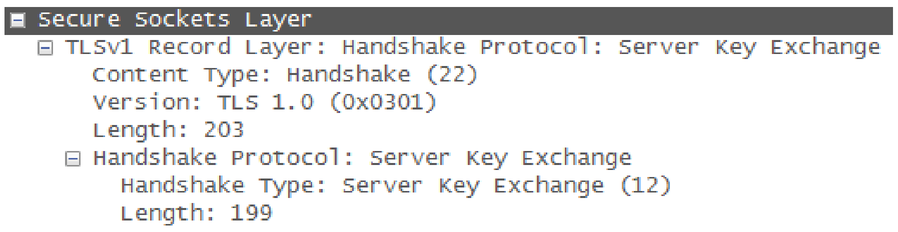
In this tab we can see the different certificates the servers send, and all their information (1608bytes). On the first line we can se the total lenght of all the certificates in the tab. we can then see the lenght of the first certificate, in the case its 910bytes. After that we see those 910bytes in data form.



```
⊟ EC Diffie-Hellman Server Params
    curve_type: named_curve (0x03)
    named_curve: secp256r1 (0x0017)
    Pubkey Length: 65
    pubkey: 049881ec625794c68ce1562a92095655d9d969adb0077df4...
    Signature Length: 128
    signature: 38967b9be69d96a342858c7013667c9e4e1a735c2568cfa1...
```

Figure 19: Certificate EC Diffie Hellman Server Params

Here the server sends it's choice out of the curves the client supported. In this case the server chose the secp256r1 curve. The server then sends the public key length, followed by the public key. This message also contains a signature that proves possession of the private key corresponding to client's public key.

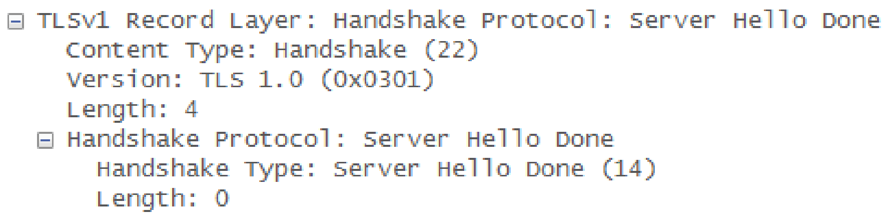


```
⊟ Secure Sockets Layer
  ⊟ TLSv1 Record Layer: Handshake Protocol: Server Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 203
    ⊟ Handshake Protocol: Server Key Exchange
        Handshake Type: Server Key Exchange (12)
        Length: 199
```

Figure 20: Certificate Server Key Exchange

The server sends it's ephemeral ECDH public key to the client

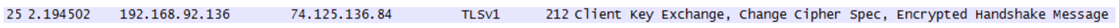
---



```
⊟ TLSv1 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 4
  ⊟ Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```

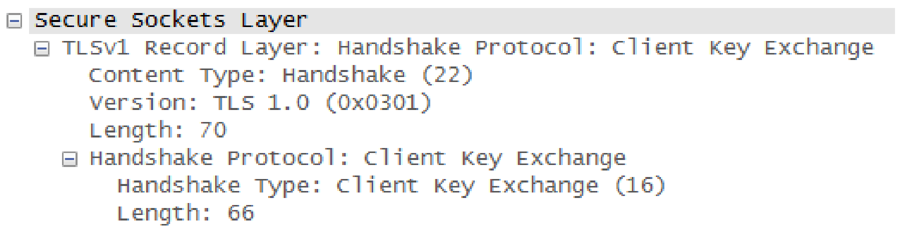
Figure 21: Server Hello Done

The server has now told the client that it is done, and are now waiting for a response from the client.



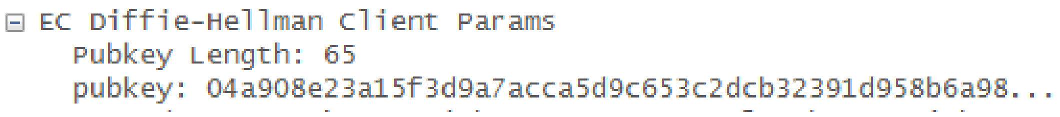
```
25 2.194502    192.168.92.136       74.125.136.84        TLSv1      212 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
```

Figure 22: Client Key Exchange



```
⊟ Secure Sockets Layer
  ⊟ TLSv1 Record Layer: Handshake Protocol: Client Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 70
    ⊟ Handshake Protocol: Client Key Exchange
        Handshake Type: Client Key Exchange (16)
        Length: 66
```

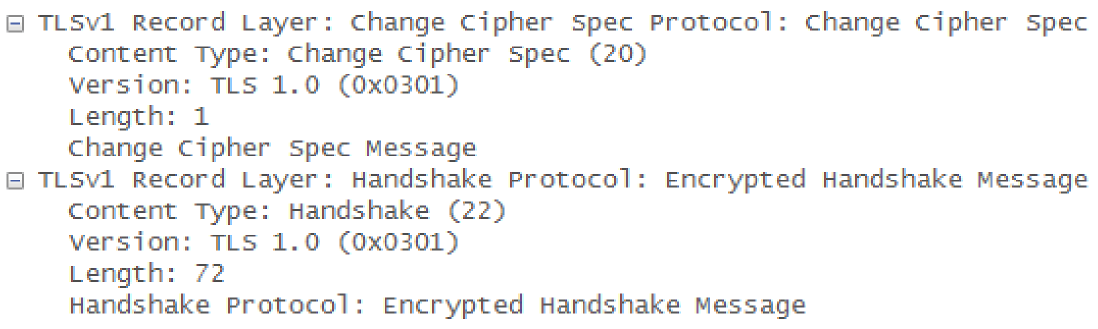
Figure 23: Client Key Exchange Version



```
⊟ EC Diffie-Hellman Client Params
    Pubkey Length: 65
    pubkey: 04a908e23a15f3d9a7acca5d9c653c2dcb32391d958b6a98...
```

Figure 24: Client Key Exchange DiffieHellman Client Params

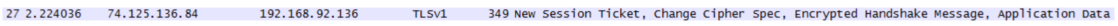
The client sends a Client Key Exchange message after computing the premaster secret using both random values. The premaster secret is encrypted by the public key from the server's certificate before being transmitted to the server. Both the client and server will compute the master secret locally and derive the session key from it.



```
⊟ TLSv1 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.0 (0x0301)
    Length: 1
    Change Cipher Spec Message
⊟ TLSv1 Record Layer: Handshake Protocol: Encrypted Handshake Message
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 72
    Handshake Protocol: Encrypted Handshake Message
```

Figure 25: Client Key Exchange Change Cipher

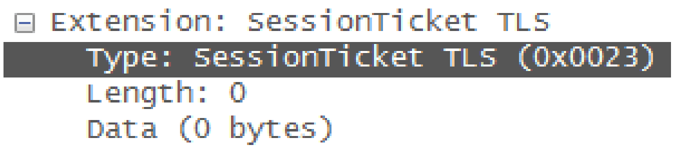
This message contains a change cipher spec message, and an encrypted handshake message wich contains a summary of the whole handshake.



```
27 2.224036    74.125.136.84         192.168.92.136        TLSv1      349 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message, Application Data
```

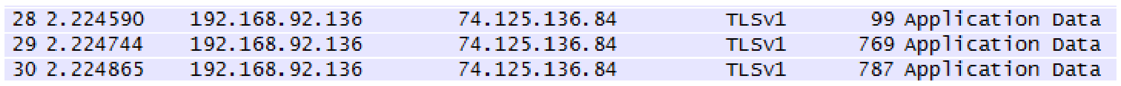
Figure 26: New Session Ticket

The client requests to start a new session after the handshake to transfer data.



```
⊟ Extension: SessionTicket TLS
    Type: SessionTicket TLS (0x0023)
    Length: 0
    Data (0 bytes)
```

Figure 27: Client Session Ticket



```
28 2.224590     192.168.92.136        74.125.136.84         TLSv1        99 Application Data
29 2.224744     192.168.92.136        74.125.136.84         TLSv1       769 Application Data
30 2.224865     192.168.92.136        74.125.136.84         TLSv1       787 Application Data
```

Figure 28: Application Data

Data packets sent after the TLS handshake is successful.s

6 Attacks Against The SSL Protocol

In the past we have seen a few different attacks targeting the SSL/TLS protocol, including BEAST [44], CRIME [45] and Lucky 13 [46].
And now, on the 20th International Workshop on Fast Software Encryption and the Blackhat Security conference in Amsterdam this year, two new attacks were presented to the participants.

What most of these attacks have in common is that they all are capable of silently decrypting browser cookies used to log in to websites. And obviously this can be a real threat against users of websites which are securing their communication with SSL/TLS.
This section will also discuss a method which is using SSL to carry out a DoS attack. This attack vector was discovered and presented back in 2011.

6.1 BEAST

In 2011, the two researchers Juliano Rizzo and Thai Duong demonstrated a POC (Proof-of-consept) called Browser Exploit Against SSL/TLS (BEAST). This attack is known as a plaintext-recovery attack.
The difference between BEAST and other similar attacks was that this exploit attacked the confidentiality of the SSL/TLS protocol.

"*During the encryption process, the protocol scrambles block after block of data using the previous encrypted block. It has long been theorized that attackers can manipulate the process to make educated guesses about the contents of the plaintext blocks.*" [?]

So if an attacker could make a correct guess, he/she would basically break the encryption and thus reveal potientially sensitive data.

Before the vulnerability was fixed, BEAST required about 2 seconds to decrypt each byte of a cookie, and since authentication cookies consists of between 1000 and 2000 characters it would take this attack about 10 minutes to work.

Needless to say, this attack was a gigantic threat against websites that used an early version of TLS, and it obviously came as a shock to people who were working in the security field at that time.

6.2 Cipher Suite Rollback

When a client starts the SSL Handshake with the server, the client sends a list over all the cipher suits it supports. The server then chooses the strongest cipher suite both parties support.
With a Cipher Suite Rollback Attack [?], an attacker can intercept the handshake, and send a list over weak and old cipher suit on behalf of the client, and thus "forcing" the client and server to use a weak encryption algorithm during the communication.
The attacker can then take advantage of this, and in the worst case scenario obtain sensitive data and/or credentials.

6.3 SSL DDoS Attack

In traditional DoS/DDoS attacks the attacker will send massive amounts of data to the server and essentially take down the server and prevent other users from accessing the content. But in 2011 a group of researchers found a more clever way to perform a DoS/DDoS attack.
When the client establish a connection to a server using a SSL connection iit requires about 15 times more processing power on the server than on the client. So by opening thousand of SSL connections the server's processor will eventually get overloaded, thus resulting in a successfull attack.

An ethical hacker that runs Darknet came up with a few different tips towards completing a successfull SSL DoS attack [?]:

- The average server can do 300 handshakes per second. This would require 10-25% of your laptops CPU
- Use multiple hosts (SSL-DOS) if an SSL Accelaretor is used.

- Don't rely on just port 443 (HTTPS). Test other ports as well; (SMTP, POP3S).

7 Graphical Representation

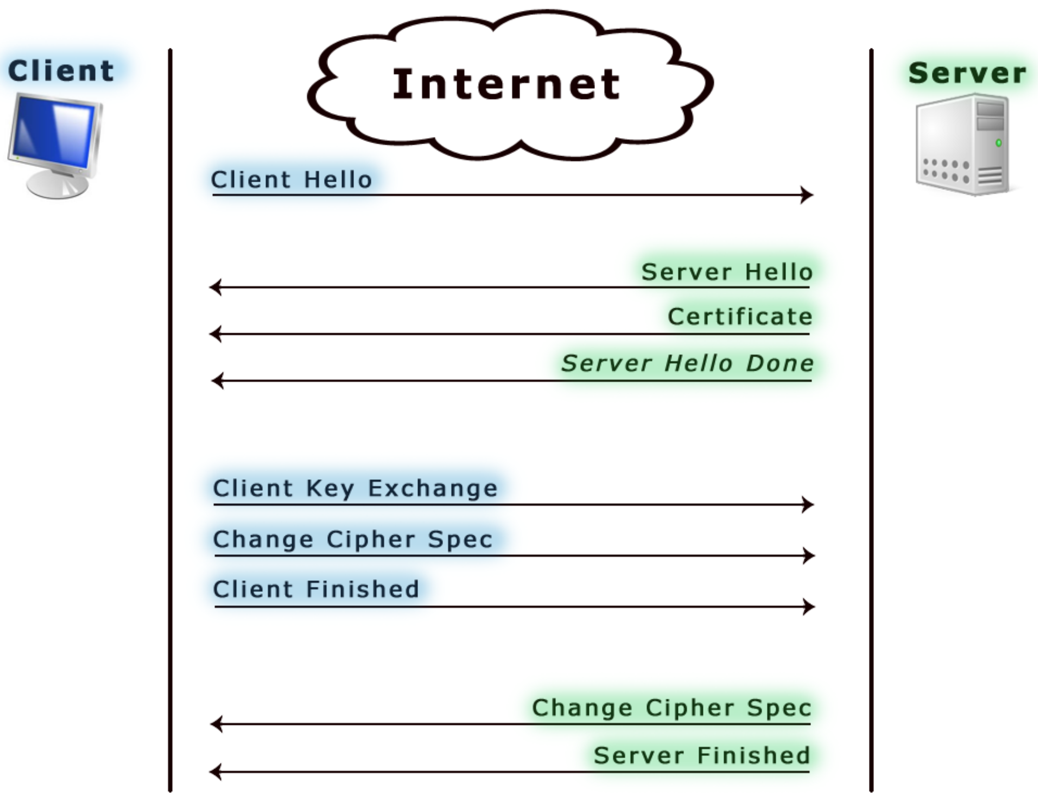


Figure 29: SSL Handshake

The graphic above shows how the SSL handshake is performed.

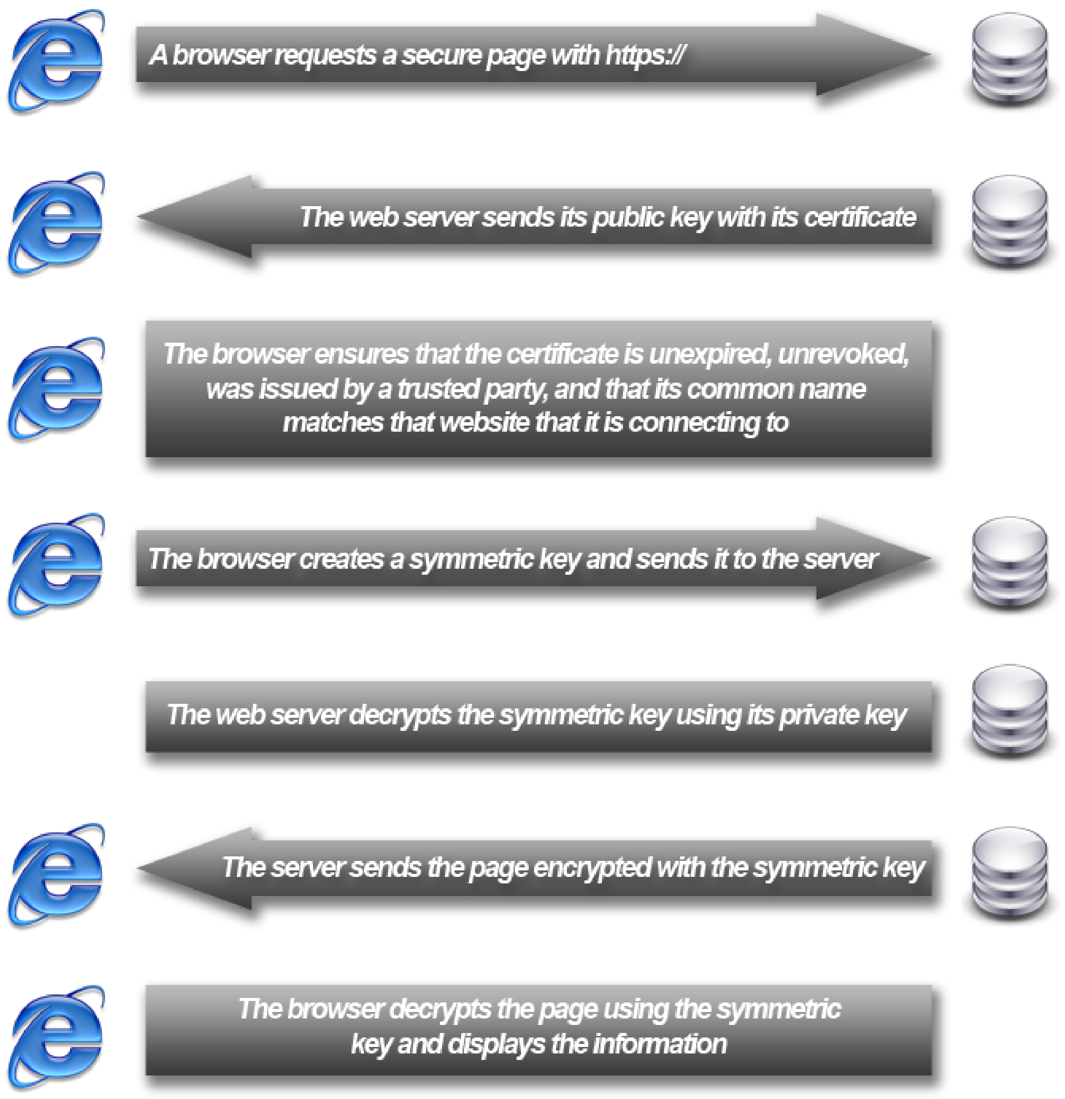

A browser requests a secure page with https://
The web server sends its public key with its certificate
The browser ensures that the certificate is unexpired, unrevoked, was issued by a trusted party, and that its common name matches that website that it is connecting to
The browser creates a symmetric key and sends it to the server
The web server decrypts the symmetric key using its private key
The server sends the page encrypted with the symmetric key
The browser decrypts the page using the symmetric key and displays the information

Figure 30: SSL connection process [7]

The graphic above discribes the SSL connection process, and how the browser communicates with the server.

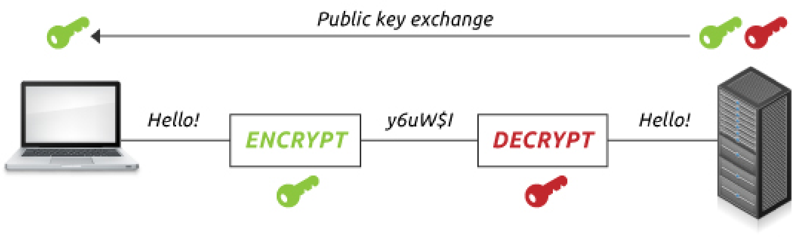


Figure 31: Asymmetric encryption [8]

Graphical description of Asymmetric encryption in a very basic way.

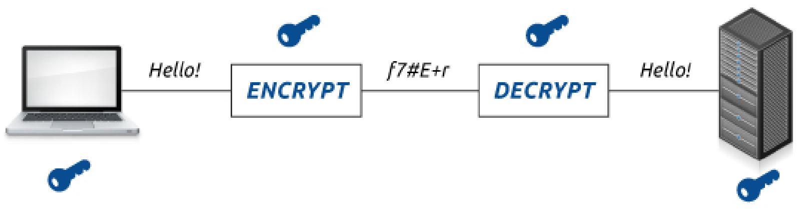


Figure 32: Symmetric encryption [9]

Graphical description of Symmetric encryption in a very basic way.

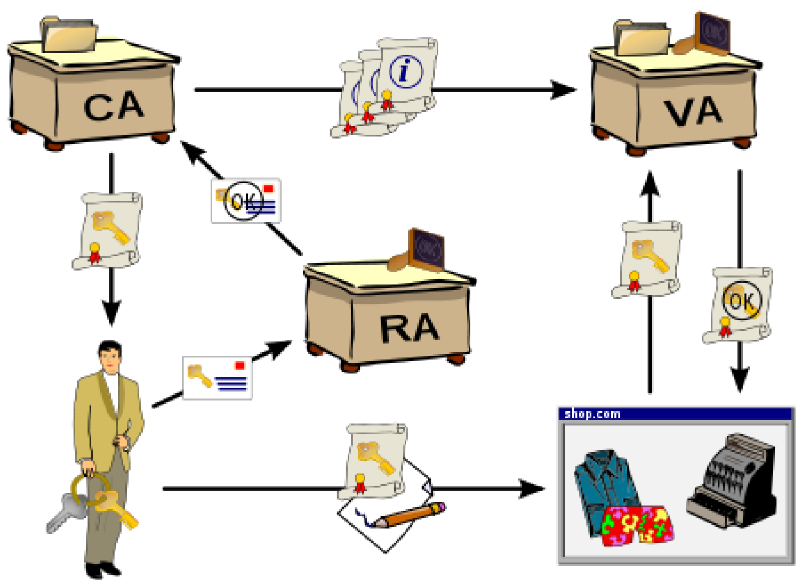
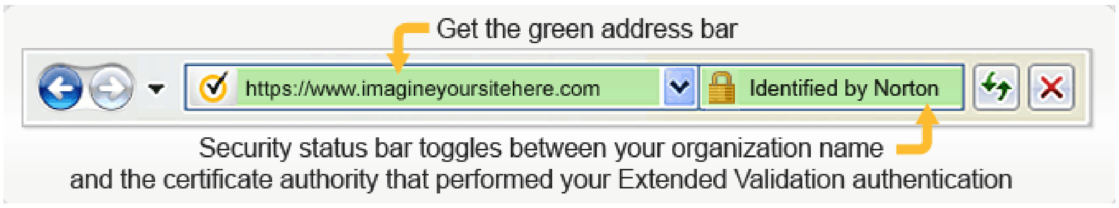


Figure 33: PKI overview [10]

This graphic shows an overview of a Public-key innfrastructure.


Get the green address bar

https://www.imagineyoursitehere.com

Identified by Norton

Security status bar toggles between your organization name
and the certificate authority that performed your Extended Validation authentication

Figure 34: Green bar overview [11]

This figure shows how some SSL certificates (often the highest trust-level) affects the users browser client.

References

[1] Wikipedia. Http secure. http://en.wikipedia.org/wiki/HTTP_Secure. Accessed 8 May, 2013.

[2] Wikipedia. Transport layer security. http://en.wikipedia.org/wiki/Secure_Sockets_Layer. Accessed 8 May, 2013.

[3] Digicert. Browser visual cues. http://edge1.digicert.com/images/ev2.jpg.

[4] Digicert. Digicert ssl description. http://www.digicert.com/ssl.htm. Accessed on 7 May, 2013.

[5] SSLShopper. The purpose of using ssl certificates. http://www.sslshopper.com/why-ssl-the-purpose-of-using-ssl-certificates.html. Accessed on 7 May, 2013.

[6] Kelly Jackson Higgins. Digital certificate authority hacked, dozens of phony digital certificates issued. http://www.darkreading.com/attacks-breaches/digital-certificate-authority-hacked-doz/231600498. Accessed May 9, 2013.

[7] SSLshopper. Ssl connection process. http://www.sslshopper.com/assets/images/what-is-ssl-connection-process.png.

[8] Digicert. Asymmetric encryption. http://edge1.digicert.com/images/encrypt-decrypt.jpg.

[9] Digicert. Symmetric encryption. http://edge1.digicert.com/images/public-key.jpg.

[10] SSLshopper. Pki overview. http://www.sslshopper.com/assets/images/pki-overview.png.

[11] Symantec. Green bar overview. http://www.symantec.com/content/en/us/enterprise/images/theme/verisign/b-thm-verisign-tab2-green-bar.gif.

[12] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - client key exchange message. https://tools.ietf.org/html/rfc5246\#page-57. Internet RFC5246.

[13] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - client key exchange message. https://tools.ietf.org/html/rfc6101\#section-5.6.7. Internet RFC6101.

[14] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - cipher suite definitions. https://tools.ietf.org/html/rfc5246\#page-83. Internet RFC5246.

[15] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - the cipher suite. https://tools.ietf.org/html/rfc5246\#page-75. Internet RFC5246.

[16] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - clienthello. https://tools.ietf.org/html/rfc5246\#page-39. Internet RFC5246.

[17] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - client hello. https://tools.ietf.org/html/rfc6101\#section-5.6.1.2. Internet RFC6101.

[18] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - serverhello. https://tools.ietf.org/html/rfc5246\#page-42. Internet RFC5246.

[19] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - server hello. https://tools.ietf.org/html/rfc6101\#section-5.6.1.3. Internet RFC6101.

[20] The transport layer security (tls) protocol - change cipher spec protocol. . Internet RFC5246.

[21] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - change cipher spec protocol. https://tools.ietf.org/html/rfc6101\#section-5.3. Internet RFC6101.

[22] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - finished. https://tools.ietf.org/html/rfc5246\#page-63. Internet RFC5246.

[23] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - finished. https://tools.ietf.org/html/rfc6101\#section-5.6.9. Internet RFC6101.

[24] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - serverhellodone. https://tools.ietf.org/html/rfc5246\#page-55. Internet RFC5246.

[25] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - server hello done. https://tools.ietf.org/html/rfc6101\#section-5.6.5. Internet RFC6101.

[26] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - server certificate. https://tools.ietf.org/html/rfc5246\#section-7.4.2. Internet RFC5246.

[27] RTFM Inc. T. Dierks, E. Rescorla. Internet x.509 public key infrastructure certificate and certificate revocation list - basic certificate fields. https://tools.ietf.org/html/rfc5280\#section-4.1. Internet RFC5280.

[28] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - the ciphersuite. https://tools.ietf.org/html/rfc6101\#appendix-A.6. Internet RFC6101.

[29] Martin E. Hellman Whitfield Diffie. New directions in cryptography. http://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf, 1976.

[30] RTFM Inc. E. Rescorla. Diffie-hellman key agreement method. http://xml2rfc.tools.ietf.org/html/rfc2631\#section-2.1, 1999. RFC2631.

[31] Inc. E. Rescorla, RTFM. Diffie-hellman key agreement method - static-static mode. http://xml2rfc.tools.ietf.org/html/rfc2631\#section-2.4. Internet RFC2631.

[32] RTFM Inc. T. Dierks, E. Rescorla. The transport layer security (tls) protocol - server key exchange message. https://tools.ietf.org/html/rfc5246\#page-51. Internet RFC5246.

[33] Jeff Forbes. Compsci 1: Principles of computer science, lecture 13. http://www.cs.duke.edu/courses/spring07/cps001/notes/lect13-4up.pdf, 2007.

[34] B. Kaliski J. Jonsson. Public-key cryptography standards (pkcs) 1: Rsa cryptography specifications version 2.1 - key types. "http://xml2rfc.tools.ietf.org/html/rfc3447\#section-3", 2003. RFC3447.

[35] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - diffie-hellman. http://tools.ietf.org/html/rfc6101\#section-6.1.2. Internet RFC6101.

[36] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - rsa. http://tools.ietf.org/html/rfc6101\#section-6.1.1. Internet RFC6101.

[37] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0 - cryptographic calculations. http://tools.ietf.org/html/rfc6101\#section-6.1. Internet RFC6101.

[38] V. Gupta C. Hawk B. Moeller S. Blake-Wilson, N. Bolyard. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls). https://tools.ietf.org/html/rfc4492. Internet RFC4492.

[39] .

[40] V. Gupta C. Hawk B. Moeller S. Blake-Wilson, N. Bolyard. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls) - supported elliptic curves extension. https://tools.ietf.org/html/rfc4492\#section-5.1.1. Internet RFC4492.

[41] V. Gupta C. Hawk B. Moeller S. Blake-Wilson, N. Bolyard. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls). https://tools.ietf.org/html/rfc4492\#section-3.1. Internet RFC4492.

[42] V. Gupta C. Hawk B. Moeller S. Blake-Wilson, N. Bolyard. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls). `https://tools.ietf.org/html/rfc4492\#section-2.3`. Internet RFC4492.

[43] RSA Laboratories. How do elliptic curve cryptosystems compare with other cryptosystems? `https://www.rsa.com/rsalabs/node.asp?id=2245`.

[44] The Register. World takes notice as ssl-chewing beast is unleashed. `http://www.theregister.co.uk/2011/09/27/beast_attacks_paypay/`. Accessed on 13 May, 2013.

[45] ArsTechnica. Crack in internet's foundation of trust allows https session hijacking. `http://arstechnica.com/security/2012/09/crime-hijacks-https-sessions/`. Accessed on 13 May, 2013.

[46] ArsTechnica. "lucky thirteen" attack snarfs cookies protected by ssl encryption. `http://arstechnica.com/security/2013/02/lucky-thirteen-attack-snarfs-cookies-protected-by-ssl-encryption`. Accessed on 13 May, 2013.