

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 13, 2013

Contents

Contents	1
1 Introduction	3
1.1 Subsection 1	3
1.2 Subsection 2	3
1.2.1 Subsubsection 1	3
1.2.2 Subsubsection 2	4
2 Short Description Of The Protocol	5
2.1 SSL Handshake	5
2.1.1 9 Steps To Secure Communication	5
3 An Overview Of SSL	7
3.1 Global Description	7
3.1.1 Advantages	7
3.1.2 Disadvantages	8
3.2 Certificate Authority	9
3.2.1 Introduction	9
3.2.2 Issuing A Certificate	9
Domain Validation	9
Example	9
3.2.3 Providers	9
3.2.4 CA Compromise	9
DigiNotar Incident	10
Consequences Of A Compromise	10
3.3 Graphical Representation	11
4 Bit-level Description	15
4.1 Handshake	15
4.2 x.509 Certificate	20
4.3 Cipher Suites	20
4.4 Authentication and Key Exchange	22
4.4.1 Diffie-Hellman	22

4.4.2	Variations on Diffie-Hellman	23
4.4.3	DSA	23
4.4.4	RSA	23
4.4.5	Variations on RSA	26
4.4.6	The Elliptic Curve variety	26
4.5	Encryption	27
4.5.1	AES - Advanced Encryption Standard	27
4.5.2	Variations on AES: AES128 and AES256	27
4.5.3	DES - Data Encryption Standard	27
4.5.4	Variations on DES - Triple-DES	27
4.5.5	RC4 - Rivest Cipher 4	27
4.6	Integrity Check	27
4.6.1	MD5 - Message-Digest algorithm 5	27
4.6.2	SHA - Secure Hash Algorithm	27
4.6.3	Variations on SHA: SHA1 and SHA256	27
5	Analysis Of The Recording	28
6	Attacks Against The SSL Protocol	31

1 Introduction

1.1 Subsection 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1.2 Subsection 2

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

1.2.1 Subsubsection 1

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit

ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.



Figure 1: Example image.

1.2.2 Subsubsection 2

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

2 Short Description Of The Protocol

2.1 SSL Handshake

When you visit a website using the https protocol [?] (*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 [?]. 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.

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.

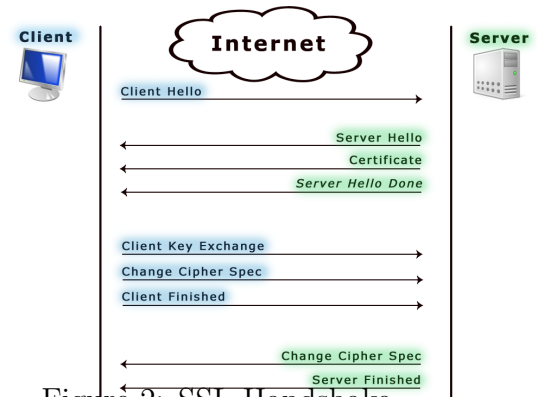


Figure 2: SSL Handshake

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 Analysis, 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

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 secured 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. [?]

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. [?]

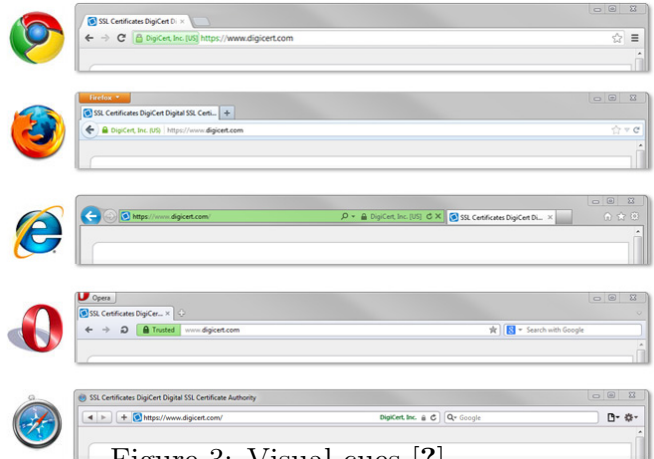


Figure 3: Visual cues [?]

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. [?]

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. [?]

3.2 Certificate Authority

3.2.1 Introduction

3.2.2 Issuing A Certificate

Domain Validation

Example

3.2.3 Providers

Worldwide there are a number of certificate authorities, 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 browsers.

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%

3.2.4 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 entity 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 [?] 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

sasasasa

3.3 Graphical Representation

Random pictures we might use:

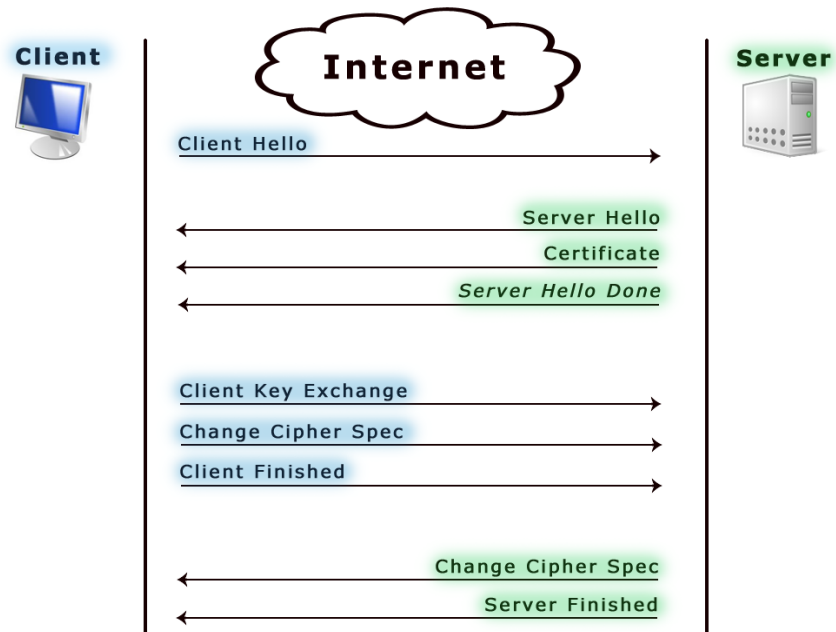


Figure 4: SSL Handshake

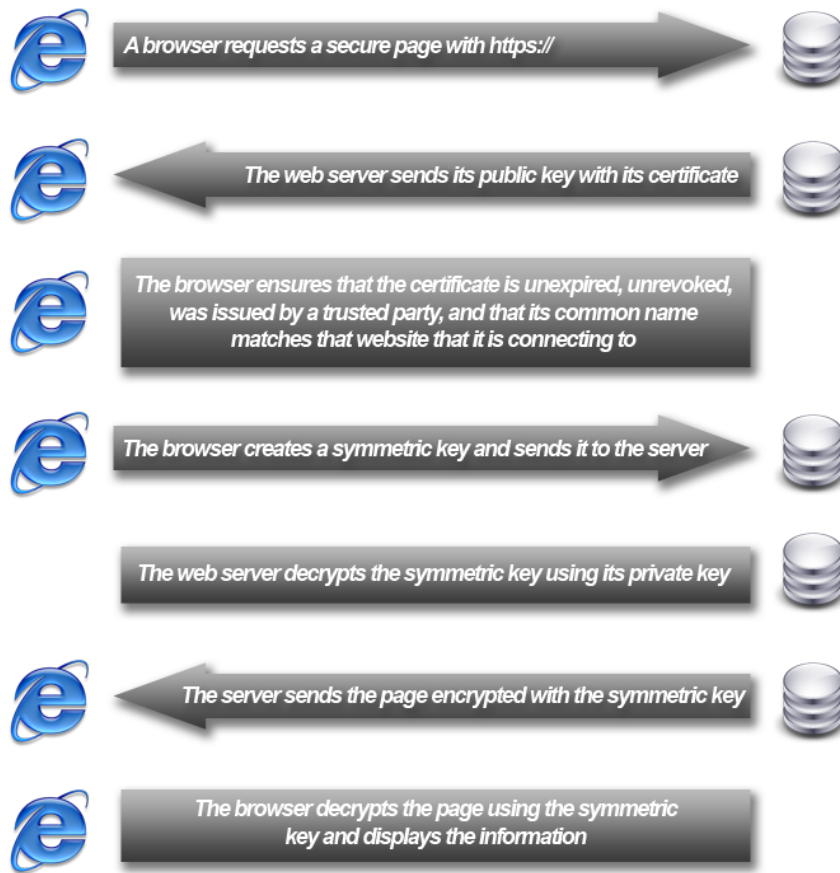


Figure 5: SSL connection process [?]

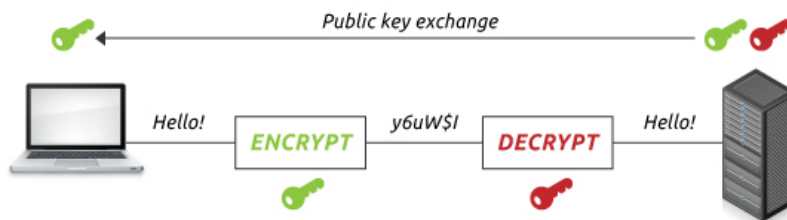


Figure 6: Asymmetric encryption [?]

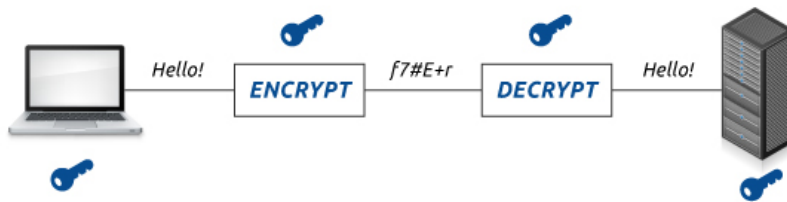


Figure 7: Symmetric encryption [?]

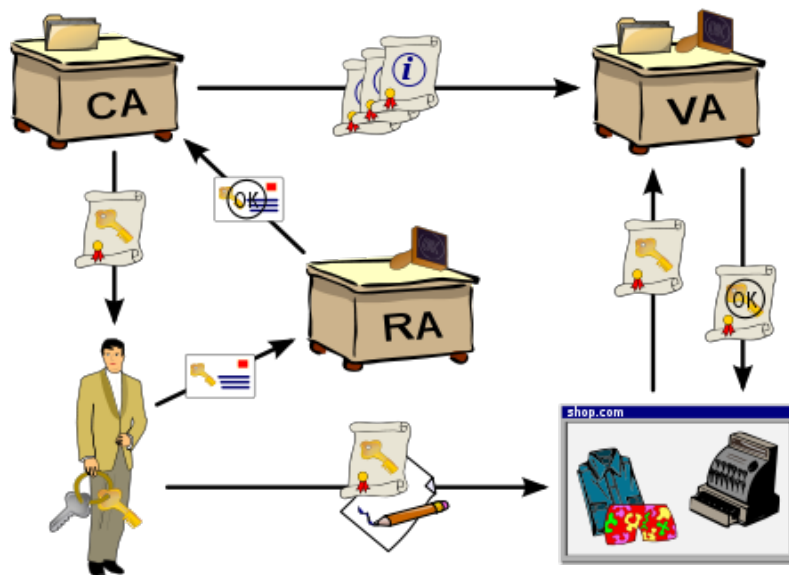


Figure 8: PKI overview [?]

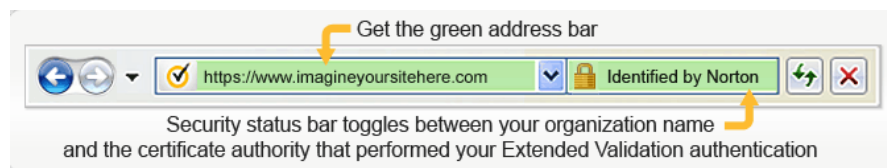


Figure 9: Green bar overview [?]

4 Bit-level Description

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* [INSERT REFERENCE], 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 [INSERT REFERENCE], 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* (message type 16). This packet contains either a *PreMasterSecret* [INSERT REFERENCE] (encrypted with the public key found in the server's certificate), a public key, or nothing at all depending on the selected protocol [INSERT REFERENCE].

The random numbers contained within the *ClientHello* and *ServerHello* 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* [INSERT REFERENCE] packet, essentially indicated that “*Now is the time to change to the cipher suite we have agreed to use, with the shared secret we calculated*”, and finally a *Finished* packet containing the *message authentication code* [INSERT REFERENCE] and hash calculated from the previous handshake messages.

Upon receiving this from the client, the server in turn responds with a similar *ChangeCipherSpec* packet 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 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. Server's 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 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 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 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 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 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 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, 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 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 of these algorithms below:

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

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. 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 generate a private integer (which must be less than the prime).

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.

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

4.4.2 Variations on Diffie-Hellman

4.4.3 DSA

4.4.4 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!

$\phi(i)$ 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 $k \pmod{\phi(M)} = 0$ would become 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 .

In the equation above, we can see that raising N to the public key produces the cipher 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 $\gcd(Q, \phi(M)) = 1$. 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.

4.4.5 Variations on RSA

4.4.6 The Elliptic Curve variety

Whilst Diffie-Hellman as mentioned above is good, it's also rather old and people are familiar with how it works. In recent years, another field of cryptography has grown increasingly popular - namely Elliptic Curve Cryptography. 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 things work slightly different. Instead, we have the domain parameters E which is the elliptic curve to use (for example p-256r1) 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 $Q * G$. 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 (<https://www.rsa.com/rsalabs/node.asp?id=2245>).

4.5 Encryption

4.5.1 AES - Advanced Encryption Standard

4.5.2 Variations on AES: AES128 and AES256

4.5.3 DES - Data Encryption Standard

4.5.4 Variations on DES - Triple-DES

4.5.5 RC4 - Rivest Cipher 4

4.6 Integrity Check

4.6.1 MD5 - Message-Digest algorithm 5

4.6.2 SHA - Secure Hash Algorithm

4.6.3 Variations on SHA: SHA1 and SHA256

5 Analysis Of The Recording

All the important parts of the recording:

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 10: Application Data

23	2.190932	74.125.136.84	192.168.92.136	TLSv1	528 Certificate
----	----------	---------------	----------------	-------	-----------------

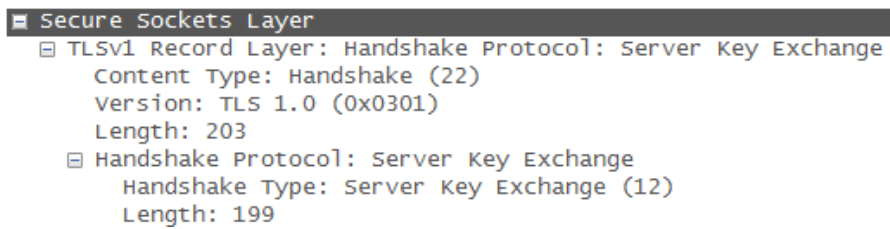
Figure 11: Certificate

```
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 12: Certificate Certificates

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

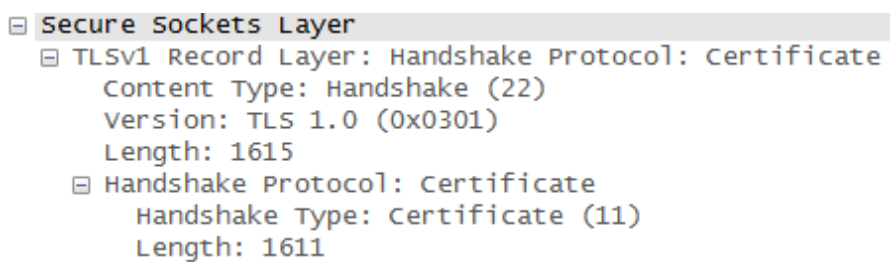
Figure 13: Certificate EC Diffie Hellman Server Params



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 14: Certificate Server Key Exchange



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 15: Certificate TLS version

```

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 16: Client Cipher Suits

```

☐ 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 17: Client Curves

```

    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 18: Client EC Point Format

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

Figure 19: Client Hello

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

Figure 20: Client Key Exchange

```

    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 21: Client Key Exchange Change Cipher


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

Figure 22: Client Key Exchange DiffieHellman Client Params

```
[-] 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

```
[-] Random
    gmt_unix_time: Feb 17, 2013 17:24:43.000000000 w. Europe Standard Time
    random_bytes: d11b597c4164265b0c0039ab58179fec0242ab1af012fdbd...
    Session ID Length: 0
```

Figure 24: Client Random

```
[-] Extension: SessionTicket TLS
    Type: SessionTicket TLS (0x0023)
    Length: 0
    Data (0 bytes)
```

Figure 25: Client Session Ticket

```
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

Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)

Figure 27: Server Cipher Suite

- ▣ 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 28: Server EC Point Formats

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

Figure 29: Server Hello

- ▣ 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 30: Server Hello Done

```
[-] Random
    gmt_unix_time: Feb 17, 2013 17:24:42.000000000 w. Europe Standard Time
    random_bytes: 7ebf304c431a8e3460c48347cf338208bf7b305522789ae2...
    Session ID Length: 0
```

Figure 31: Server Random

```
[-] 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 32: Server TLS Version

```
[-] 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 33: TLS version

6 Attacks Against The SSL Protocol

In the past we have seen a few different attacks targeting the SSL/TLS protocol, including BEAST [?], CRIME [?] and Lucky 13 [?].

And now, on the 20th International Workshop on Fast Software Encryption and the Blackhat Security conference in Amsterdam this year, these new attacks were presented to the participants.

What 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 may be a real threat against users of websites which are securing their communication with SSL/TLS.