A Network Security Spotlight on SSL/TLS and HTTPS

A Student Presentation
(*class notes*)
CIS 6650 Computer Security

Eddie Yee Tak Ma
http://eddiema.ca

School of Computer Science
University of Guelph

Posted to the class on March 21st, 2011
Posted online on February 4th, 2012

Instructor:
Dr. Charlie Obimbo

All figures herein were illustrated by Eddie, and are expressly released to the public domain.

1 Preamble

This document is designed for students or professionals with some working knowledge of network communication, computer science and computer security. After reading this presentation or attending the accompanying presentation, the reader should have a general grasp of the TLS (and HTTPS) specifications and what they have to offer. Moreover, the reader will know who the Internet Engineering Task Force (IETF) is and where to look up the complete specifications published by the IETF – useful for (1) diagnosing bugs within, or (2) deploying a new implementation.

2 Overview

When machines must communicate over a network, vulnerabilities exist which permit attacks to occur. In this document, we discuss what security is conferred for such communications when a protocol such as TLS is used.

In this presentation, we will first overview the Internet Protocol (IP) stack, then focus in on the Transport Layer and Application Layer. We then describe a hypothetical application using TLS and overview a typical session that such a client and server software will experience. Finally, we stipulate the arrangement of data into *Records* when using TLS and also the accepted forms for which we find certificates used for authentication in a widespread application of TLS called HTTPS.

3 Layers upon Layers (*Architecture*)

Let's review the Internet Protocol (IP) layers we use in day-to-day network communication. This will allow us to discuss the *where* within the architecture we can evesdrop or secure communications. Figure 1 is a schematic of the IP stack.

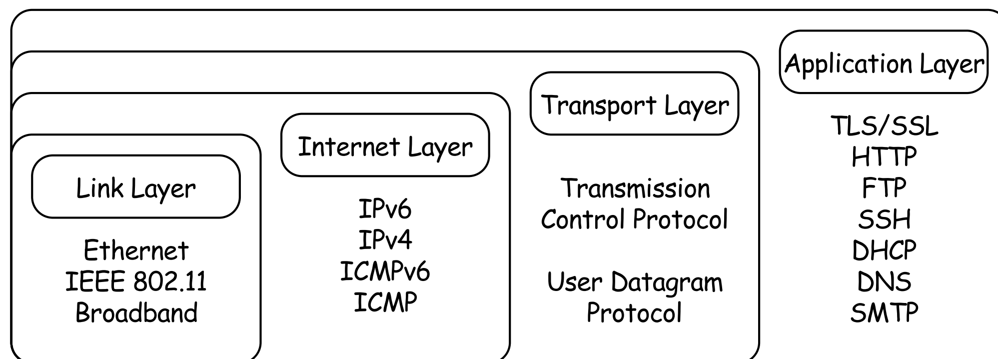

Figure 1: A schematic of the Internet Protocol (IP) stack.

As we move from the link layer to the application layer, the protocols become increasingly abstract and specifically targetted toward a particular use. Note that the principal of data encapsulation stands so that a more abstract layer *must never* have any knowledge of its underlying layers. The *Link Layer* represents the physical connections between machines *e.g. IEEE802.11 (a family of wireless connections)*. The *Internet Layer* consists of a family of protocols that can

create the elementary software channels between machines *e.g. IPv6 (Internet Protocol version 6)* or diagnose and report on said channels *e.g. ICMPv6 (Internet Control Message Protocol)*. At the *Transport Layer*, protocols exist which define the form in which data must be arranged in order to be transceived. Finally, the *Application Layer* defines protocols that allow end-user (or system) application software to communicate *e.g. HTTP (hypertext transfer protocol)* for browsers and *e.g. TLS/SSL (transport layer security/secure sockets layer)*[1] for encrypted and/or authenticated communications.

3.1 Being Eve

Having identified the layers in our IP stack, we now have a frame of reference for discussing where an attacker could potentially invade, alter or disrupt the flow of communications. Recall that we should think of security in terms of *Confidentiality*, *Integrity* and *Availability*.

To compromise confidentiality or integrity, an attacker must access a network via an intercepted physical connection between machines. The difficulty of this task varies with the communication medium. As you are likely aware, the IEEE802.11 family of wireless connections is particularly vulnerable especially when no encryption (or weak encryption) is used. Once access is obtained, the attacker can begin to read the transmitted data or to introduce new data. Finally, availability can be compromised by severing existing physical connections or by flooding machines with irrelevant data as in a Denial of Service (DoS) attack.

3.2 Network Security in the Application Layer

In the present discussion, we will focus on the security that can be provided at the *Application Layer* of the IP stack. It is here that we find TLS, HTTP and also HTTPS. Importantly, we cannot mitigate attacks with respect to availability. For example, should a hardware connection be severed or interrupted by an attacker, we are clearly working on a level that is too abstract to help.

We are however capable of increasing the security of our transcieved data in the respects of confidentiality and integrity as TLS and HTTPS are each amenable to encryption and to a lesser degree, authentication.

4 HTTPS and S-HTTP are not the same.

First, an aside; we must clarify the difference between HTTPS [4] and S-HTTP [3]. HTTPS and S-HTTP can both be considered an implementation of HTTP with some security built-in (that is, authentication and encryption). Both of these protocols were described and appeared in the 1990's; however, only HTTPS enjoyed adoption by both Microsoft and Netscape leading to its current widespread use. HTTPS (*hypertext transfer protocol secure*) is made secure by virtue of being a deployment of HTTP over TLS while by stark contrast, S-HTTP (*secure hypertext transfer protocol*) is made secure by its own protocols that extend the HTTP definition. It should be noted that because S-HTTP never gained popularity, that it is often mistaken for HTTPS.

The items TLS (and HTTPS), and S-HTTP are all documented by The Internet Engineering Task Force (IETF) [1]. The documents that describe standards and experimental items exists as a collection of memos indexed by serial number.

---

[1] Transport Layer Security (TLS) is the successor to Secure Socket Layer (SSL).

Let us now discuss first TLS, then HTTPS.

5 Transport Layer Security (TLS)

The TLS defines a set of communication and security guidelines for a session-based connection between a client and a server. Being meant to be deployed in the application layer, TLS is implemented on top of the transport layer of the IP stack. TLS is not specific to a particular transport protocol and can layered on top of anything that is deemed reliable by the implementer (*i.e. you may choose your own adventure*). For instance, one is likely to use TCP/IP or a variant thereof but not UDP. The looseness of this definition is owed to the fact that TLS as described is a protocol (a set of agreed upon behaviours) – similar to an API (application program interface). Clients and servers implementing TLS are expected to connect in a specific way (*shake hands, bow*) communicate with certain guidelines (*no foul language*) and part with sufficient notification (*excusing oneself politely and not slamming the door*). In contrast, it is not a specific implementation (or even implementation specific!) and makes no assumptions about programming languages or operating systems and allows the client and server to negotiate on message authentication and encryption. All of this can be defined (or at least greatly influenced) at the leisure of the implementer. The specification is abstracted at exactly a level that allows it to be both (1) versatile enough to run ontop of a variety of transport protocols and (2) precise enough that stipulations about the arrangement of data are guaranteed to be homogeneous.

Next, we need to discuss the idea of CipherSuites.

5.1 Introducing CipherSuites

The CipherSuite is the component of TLS which determines the strength of the security of a particular TLS session. A CipherSuite is a named four-tuple of components which together are used for the authentication and encryption of a session. In reality, this is communicated between a client and a server as an integer and often decoded as some enumeration. These components are (1) a key exchange algorithm, (2) an encryption algorithm, (3) a message authentication code (MAC)[2], (4) a pseudorandom function (PRF).

The key exchange algorithm determines how authentication should occur during handshaking. The bulk encryption algorithm describes how messages are to be encrypted (*block ciphers*). The MAC creates a cryptographic hash of each block in the message stream. Finally, the PRF creates a 48-byte *master secret* which is then used by the client and server to create session keys.

As of this writing, the latest complete standard of TLS is described in RFC 5246 [6]. Because the TLS definition is very extensive, we will go into detail for only the creation of a connection, the permitted format of messages and the termination of a connection under normal circumstances. Please see RFC 5246 [6] if you are interested about exceptional circumstances.

For the purposes of this discussion, let us assume to discuss a hypothetical implementation of TLS over TCP/IP.

[2]Not to be confused with *Mandatory Access Control* in database security theory.

5.2 Handshaking

In a client-server arrangement, machines that are designated as servers allow specific software to wait for incoming connections. Specifically, this software is restricted to a well defined set of ports and transport protocols. We say that this software is listening for an incoming request for a connection on such a given port on this machine. For TCP specifically, this software is said to *bind* with the port and the port is said to be *passive open*. The purpose of a piece of software listening on a port is so a connection can be made when a request to do so is heard.

A machine that requests a connection from a server is called a client.

Because our implementation runs ontop of TCP, all of the leg work that must be done by TCP must be completed before anything more abstract using TLS can begin. Our client thus commits the three-way handshake expected in the TCP protocol (please see IETF's RFC 703 [2] if you would like more details about TCP – suffice it to say that knowing *that a handshake occurs* is sufficient for our discussion here).

We assume that the handshake of TCP (our chosen transport layer) has completed normally; i.e. we assume a valid TCP connection exists between the client and the server. Note that while data is arranged in *packets* when sent via TCP – data is further arranged in *records* when sent via TLS.

We can now discuss how a *Simple TLS handshake* must proceed.

5.2.1 Step 1. Negotiation

The client sends a *ClientHello* to the server. The *ClientHello* is a record that contains the following information: the protocol versions of TLS/SSL supported by the client; the CipherSuites that the client supports; the compression methods supported and a randomly generated value. The client may also provide a session ID if it is trying to resume a previous session with the server.

The server sends back an *ServerHello* to the client. The *ServerHello* is a record that contains: the highest protocol version supported by both the client and the server; a single CipherSuite supported by both; a single compression method support by both and a random value. The server can send back the same session ID provided by the client if the server is able to resume the named previous session.

The client and server random values are used to calculate several keys during this session.

The CipherSuite agreed upon by the client and the server indicates how authentication should occur at this stage of the handshaking. If the CipherSuite supports it, the server sends a *Certificate* message to the client. This certificate must be of the type X.509v3 (unless otherwise negotiated).

The server then sends a *ServerHelloDone* to indicate the conclusion of this negotiation.

The client replies with a *ClientKeyExchange* record that can contain a *PreMasterSecret*, a public key or nothing depending on the CipherSuite chosen. For our hypothetical implementation, we will assume that a *PreMasterSecret* is in the reply. The *PreMasterSecret* along with some random numbers are used to create the 48-byte *MasterSecret*. The *MasterSecret* is independently calculated for each the client and the server but is identical. The *MasterSecret* is used as the source of entropy in the creation of the remaining keys used in this session.

5.2.2 Step 2. The Server checks the Client's Work

The client sends a *ChangeCipherSpec* record to the server. All records after this one will be authenticated and encrypted given the CipherSuite and MasterSecret chosen. An encrypted *Finished* message is then sent by the client containing a hash and MAC of the previous handshake messages. The server decrypts the *Finished* record and ensures that the hash and MAC are correct. If the server does not recover the same solution as the client, then handshaking has failed the TLS must be terminated (the application may also choose to terminate the below TCP connection).

5.2.3 Step 3. The Client checks the Server's Work

The server sends a *ChangeCipherSpec* record followed with an encrypted *Finished* record. The client checks that the hash and MAC contained therein can be decrypted with the chosen CipherSuite to ensure that correct encryption was performed. If the client cannot recover the same solution, the connection must be terminated.

5.2.4 Step 4. Exchanging Application Records

Since the handshake is complete, application records can now be exchanged. Authentication and encryption of application records is provided by the CipherSuite given. In our hypothetical implementation, the actual useful application data is sent at this point (*e.g. the streaming of media, the serving of realtime remote sensing data, or the serving of static files*).

Figure 2 summarizes the above discussion on the *Simple TLS Handshake* schematically.

5.3 Modification: Authenticating the Client as well

The above described procedure outlines the handshaking performed in TLS when only the server is authenticated with a certificate (*Simple TLS handshake*). When the client also requires authentication, the negotiation is changed so that the client sends a *Certificate* message after receiving *ServerHelloDone* and a *CertificateVerify* message after sending *ClientKeyExchange*. The client's certificate consists of a public key and private key pair; the *CertificateVerify* message is an encryption of a signature on all previous handshaking messages using the client's private key. The server is able to authenticate the client by using the published public key.

5.4 A Little More on CipherSuites

Up until now, we have been very abstract about CipherSuites. Now that we have been given some insight into their use, we can delve a bit deeper into the CipherSuite.

The authentication algorithms that the CipherSuite may define include Rivest-Shamir-Adleman (RSA) and Diffie-Hellman (DH); both of which require the use of prime numbers in modulus exponentiation. Encryption can be performed with Rivest Cipher 4 (RC4), Triple Data Encryption Standard (3DES), and Advanced Encryption Standard (AES). You'll notice that these encryption algorithms are all block ciphers, rather than the prime exponentiation functions above. Block ciphers are chosen because they are less computationally intensive. The MAC (essentially a hashing function to test message integrity) can be performed with Message-Digest algorithm 5 (MD5) and Secure Hash Algorithm (SHA).

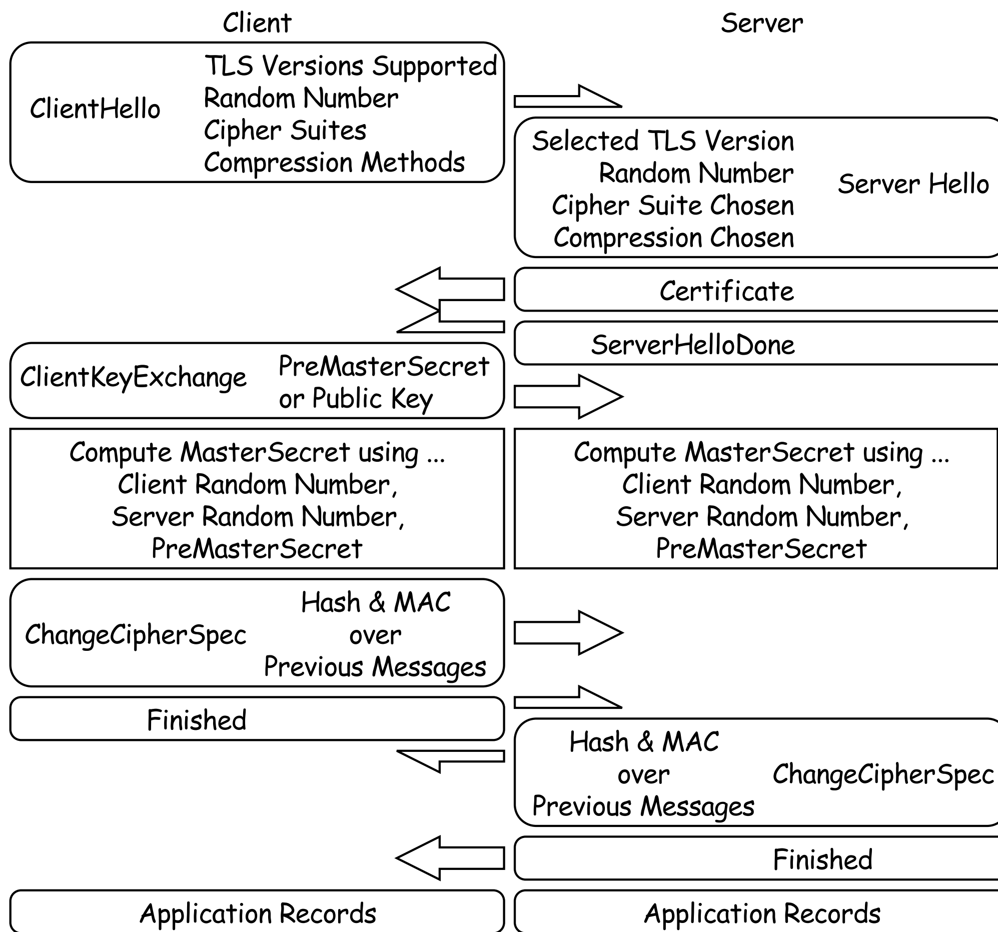

Figure 2: A schematic of the handshaking procedure.

The listing below is an extraction from RFC 5246 [6] which shows a list of the possible CipherSuites supported in TLS v1.2.

Cipher Suite	Key Exchange	Cipher	Mac
TLS_NULL_WITH_NULL_NULL	NULL	NULL	NULL
TLS_RSA_WITH_NULL_MD5	RSA	NULL	MD5
TLS_RSA_WITH_NULL_SHA	RSA	NULL	SHA
TLS_RSA_WITH_NULL_SHA256	RSA	NULL	SHA256
TLS_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	AES_128_CBC	SHA
TLS_RSA_WITH_AES_256_CBC_SHA	RSA	AES_256_CBC	SHA
TLS_RSA_WITH_AES_128_CBC_SHA256	RSA	AES_128_CBC	SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256	RSA	AES_256_CBC	SHA256
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES_EDE_CBC	SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA
TLS_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4_128	MD5
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA	DH_DSS	AES_128_CBC	SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA	DH_RSA	AES_128_CBC	SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	DHE_DSS	AES_128_CBC	SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	DHE_RSA	AES_128_CBC	SHA
TLS_DH_anon_WITH_AES_128_CBC_SHA	DH_anon	AES_128_CBC	SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA	DH_DSS	AES_256_CBC	SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA	DH_RSA	AES_256_CBC	SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	DHE_DSS	AES_256_CBC	SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	DHE_RSA	AES_256_CBC	SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA	DH_anon	AES_256_CBC	SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	DH_DSS	AES_128_CBC	SHA256
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	DH_RSA	AES_128_CBC	SHA256
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	DHE_DSS	AES_128_CBC	SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	DHE_RSA	AES_128_CBC	SHA256
TLS_DH_anon_WITH_AES_128_CBC_SHA256	DH_anon	AES_128_CBC	SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	DH_DSS	AES_256_CBC	SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	DH_RSA	AES_256_CBC	SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	DHE_DSS	AES_256_CBC	SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	DHE_RSA	AES_256_CBC	SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256	DH_anon	AES_256_CBC	SHA256

5.5 The Record Protocol

Having succeeded in handshaking, the client and server can now exchange data using their agreed upon encryption method expressed in the selected CipherSuite. The application messages are sent in the Record Protocol format. Figure 3 is a schematic that indicates the arrangement of data in a record protocol.

+	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	Content type			
Bytes 1..4	Version		Length	
	Major	Minor	bits 15..8	bits 7..0
Bytes 5..(m-1)	Protocol message			
Bytes m..(p-1)	MAC (optional depending on CipherSuite)			
Bytes p..(q-1)	Padding (only if the CipherSuite defines a block cipher)			

Figure 3: A schematic of the data arranged in a TLS record.

The *Content type* field is a single byte that may take the values 22 (0x16) for handshake, 20 (0x14) for *ChangeCipherSpec*, 23 (0x17) for application or 21 (0x15) for alert (exceptional or terminational circumstances). The version (3, 0) corresponds to SSL 3.0; versions (3, [1..3]) correspond to TLS versions 1.[0..2] respectively. The length indicates the protocol message length only (i.e. it does not include the number of bytes in the MAC or block cipher padding). Notice that encryption of the message and MAC is handled exclusively by TLS and the application logic should not have any knowledge of it. Figure 4 is a schematic of a record in our fictitious protocol.

+	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	0x17	Content type is "Application"		
Bytes 1..4	Version		Length (30_{10} = 0x0116)	
	0x3	0x3	0x1	0x16
Bytes 5..(m-1)	"<FOOD>WHERE'S THE FOOD?</FOOD>" (30 Bytes excluding '\0')			
Bytes m..(p-1)	0x C402 4EFA AAD2 3EE3 83A4 0822 1182 DD77 (128-bit MD5)			
Bytes p..(q-1)	(no block cipher in fictitious example)			

Figure 4: A schematic of a record for our fictitious application – in real life, TLS will encrypt the message.

Notice there can never actually be a record like this in real life as this record has neither

authentication nor encryption but only a MAC in the form of a 128-bit MD5. In real life, the text would be encrypted with a block cypher and padding may be introduced to satisfy the number of bits the block cypher requires to work.

5.6 Closing the Connection

Both the server and the client are able to signal to the other that the session should end. The closing message increases the difficulty in performing a truncation attack. The message *CloseNotify* is sent. The sender must terminate the write side of the connection immediately after sending *CloseNotify*. A recipient of *CloseNotify* must immediately send back a *CloseNotify*. The sender does not need to wait for a *CloseNotify* reply before terminating the read side of the connection. In our hypothetical implementation, if the TCP connection needs to persist after the TLS layer is closed, then the sender must wait for a *CloseNotify*. Otherwise, should the underlying transport layer be closed immediately anyway, then there is no need to wait.

6 HTTPS

Now that we are aware of what TLS has to offer in terms of authentication (at least the server is known and verified with a public key), encryption (the data has been scrambled with block ciphers), and integrity (a hash function is used at the end of each block); we can discuss a practical application we are all familiar with – HTTP. If we transmit HTTP on top of TLS as the application data (using Application Records), the result is HTTPS [4]. This means that we use TLS to wrap all of the messages sent e.g. from a webserver to a webbrowser.

Let us take a moment to appreciate this. When you visit a URL with the HTTPS protocol, your browser first negotiates a connection on port 443 (instead of port 80 for plain HTTP) with the server by passing along a *ClientHello*. The two exchange random numbers, a list of supported CipherSuites and compression schemes, and then the client retrieves a certificate from the server for authentication. Then with an agreed upon CipherSuite, the client receives all of the data requested and decrypts it with the best supported encryption block cypher that the writers of your browser software implemented. All of the bytes have been received and decrypted and the page is cached and rendered in your browser. The HTTPS connection is terminated; the underlying TLS connection is terminated; the underlying TCP transport is terminated.

The only thing missing is the authentication of the certificate.

The certificate is considered authentic by a browser when the writer of its software trusts a certificate authority who trusts the server with the certificate. This means that you have delegated your own decision to trust a site to both the browser authors and the certificate authority.

Human error and malice notwithstanding, our discussion proceeds where all parties are trustworthy.

The public key certificate provided for authentication during handshaking contains a hostname (domain name[3]). This hostname is checked against the URI (URL[4]) that should belong to the server – when these two values are not the same, then something is wrong (perhaps a few bits were flipped during transport). A connection is still permitted where authentication has failed but

[3]For our purposes, a hostname and a domain name can considered be the same.

[4]For our purposes, a URI and a URL can also be considered the same.

encryption still persists. In this case, the browser software must present the user with notification that the certificate is not correct given the server at the specified URL. A widely used certificate format is X.509 [5]. A schematic of this certificate is shown in Figure 5.

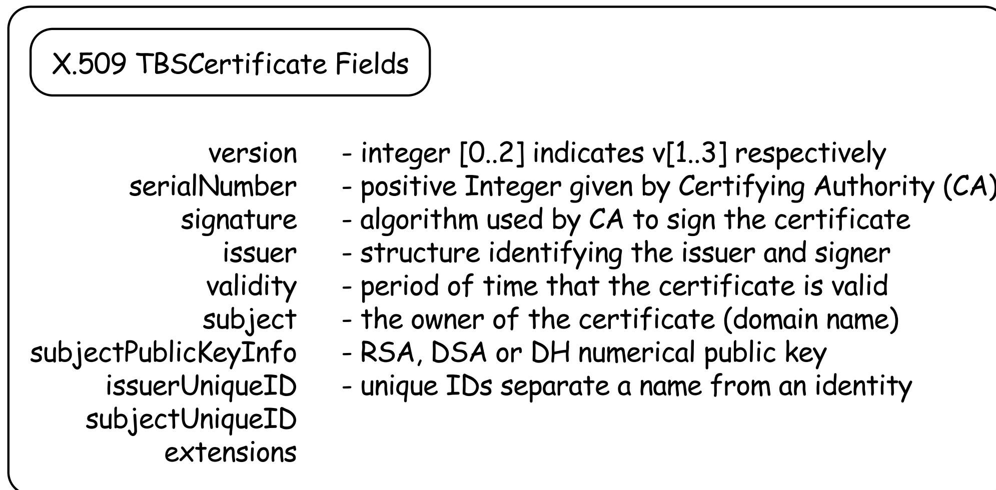

Figure 5: A schematic of the data arranged in a certificate transmitted during handshaking.

Notice that the Public Key is used for encryption using the number theory of large primes and modulus exponentiation as we have seen before.

7 Conclusion

Having read this document or attended the accompanying talk, you should now have a general understanding of how SSL/HTTPS improves security for HTTP. We have explained the components of a simple handshake and indicated what is changed when the client is also authenticated. We have described the very important CipherSuite and also the formats of the record and certificates used during communication. Finally, you have been made aware of the IETF and where to find the relevant specification documents.

References

[1] The internet engineering task force (ietf). http://www.ietf.org (accessed March 2011).

[2] University of Southern California Information Sciences Institute. Rfc 793 transmission control protocol functional specification, September 1981. http://tools.ietf.org/html/rfc793 (accessed March 2011).

[3] The Internet Society. The secure hypertext transfer protocol, AUGUST 1999. http://tools.ietf.org/html/rfc2660 (accessed March 2011).

[4] The Internet Society. Rfc 2818 http over tls, May 2000. http://tools.ietf.org/html/rfc2818 (accessed March 2011).

[5] The IETF Trust. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile, May 2008. http://tools.ietf.org/html/rfc5280 (accessed March 2011).

[6] The IETF Trust. Rfc 5246 the transport layer security (tls) protocol version 1.2, August 2008. http://tools.ietf.org/html/rfc5246 (accessed March 2011).