TLS Extensions

Project

IMT4101 - Network Security

Spring 2004

Ole Martin Dahl [ole.dahl@hig.no]
Torkjel Søndrol [torkjel.soendrol@hig.no]
Fredrik Skarderud [fredrik.skarderud@hig.no]
Ole Kasper Olsen [ole.olsen@hig.no]

Abstract

This report will focus on extensions for TLS. We will start with a quick overview of the TLS protocol in section 1. We will here present the basics of the protocol, before we advance to a description of the TLS extensions which are proposed in the Internet community at the time of writing in section 2. Thus, section 1 will not contain a very detailed description of the TLS protocol and its messages, only enough to make a common base for further discussion about TLS extensions. Section 3 will round off the report with our own suggestions and vision for the future of TLS.

Contents

1 SSL and TLS 3
1.1 Introduction and the Goal of TLS 3
1.2 The Record Layer Protocol 3
1.3 The Handshake Protocol 4
1.3.1 Setting up Encrypted Communication 4
1.3.2 Server Authentication 5
1.3.3 Mutual Authentication 6
1.3.4 Resuming a Session 7
1.4 Other Protocols in TLS 7
1.4.1 Change Cipher Spec Protocol 8
1.4.2 Alert Protocol 8
1.4.3 Application Data Protocol 8
1.5 Pseudorandom Function 8
1.6 Current Cipher Suites in TLS 9
1.6.1 General Cipher Suites 9
1.6.2 Kerberos Cipher Suite 9
1.6.3 Cipher Suites and Perfect Forward Secrecy 9
2 TLS Extensions 10
2.1 Extended TLS Handshake 11
2.1.1 Server Name Indication 11
2.1.2 Maximum Fragment Length Negotiation 12
2.1.3 Client Certificate URLs 12
2.1.4 Trusted CA Indication 13
2.1.5 Truncated HMAC 13
2.1.6 Certificate Status Request 13
2.2 Future Cipher Suites Based on Extensions 13
2.2.1 OpenPGP Keys for TLS Authentication 13
2.2.2 Elliptic Curve Cryptography for TLS 14
2.2.3 Secure Remote Passwords for TLS Authentication 15
2.2.4 Shared Keys in TLS 16
3 TLS' Future 17
3.1 TLS version 1.1 17
3.2 The Future of TLS 17
3.3 Visions 18
3.3.1 Biometric Authentication Extension 18
4 References 20

1 SSL and TLS

1.1 Introduction and the Goal of TLS

SSL was developed by Netscape in 1994 to provide secure communications between Web clients (browsers) and Web servers. It relies on certificates and public key cryptography for symmetric key exchange and authentication. TLS is in essence a standardised version of SSL v3.0, although TLS has enough minor additions and differences for the two versions to be incompatible. TLS may back down to a SSL v3.0 feature set if the client doesn't support TLS.

TLS provides an entirely new protocol layer between the transport layer and the application layer, dedicated for security. As a side effect, TLS may provide security for other applications than just HTTP. Providing a new protocol layer will not make the existence of SSL/TLS encryption entirely transparent to the user, as it is with for example IPSec where security is integrated with a core protocol layer (Internet Protocol), and support for TLS needs to be specially programmed into either the software application (i.e. the application layer), or it may be implemented as a part of the underlying protocol suite (TCP transport layer) for a more general solution. Integration into specific applications is by far the most common approach, like in all modern Web browsers.

TLS consists of several protocols, often conceptually visualised in two layers. The *record layer protocol* acts as a bottom layer which encompasses all of the other protocols, of which the Handshake protocol is arguably the most important one.

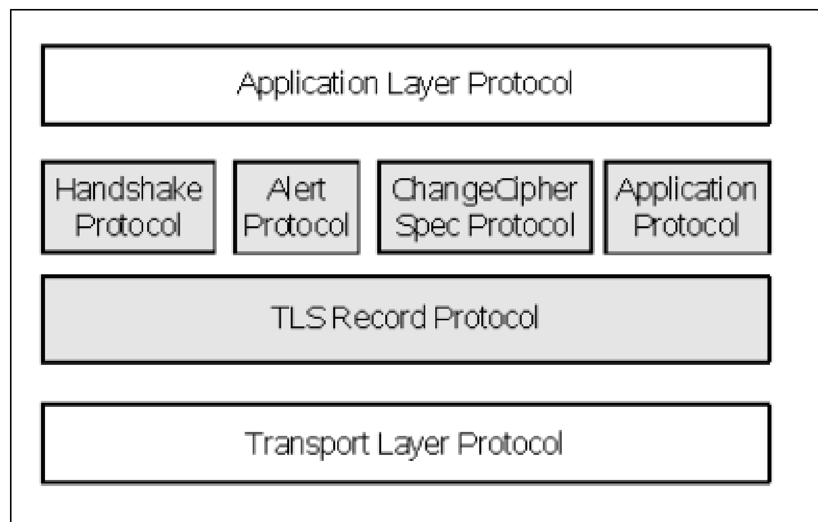


Figure 1: TLS Protocols

1.2 The Record Layer Protocol

TLS uses a *record layer* to encapsulate all data. It accepts all of the different messages the other TLS protocols send and wraps them in a *record layer header* before sending the packet further down to the transport layer.

The record layer's main task is to provide a frame for the other TLS protocols and to secure application data. It does this by wrapping the TLS message in a 5-byte header which declares what type of TLS protocol message is in the package, which version of SSL or TLS is used (versions 3.0 and lower are SSL, version 3.1 is TLS) and how long the message is. The record layer can then add a *message authentication code* (MAC) to the message, then compress and encrypt it if required.

The TLS specification opens for use of one of two message digest algorithms when generating the MAC, namely Rivest's MD5 [RFC 1321] and the US Government's SHA-1 [FIPS PUB 180-

2]. The MAC is appended to the application data, and then the whole message including the MAC is encrypted. The MAC also protects, in addition to the application data, the record layer header in the TLS message which contains TLS version numbering, message length and more, even though that part of the message is not encrypted.

The MAC is not simply a hash value of the message. The TLS specification requires the use of *HMAC* (*Hashed Message Authentication Code*) [RFC 2104], which details a specific algorithm in which to use the chosen hash function several times along with a secret key. The HMAC algorithm has been carefully scrutinised by the cryptographic community and is considered to be a very good way to use hash functions to create a good cryptographically secure MAC.

Note that the SSL version 3.0 specification does not specify the HMAC for use as a MAC, but rather a MAC based on the HMAC draft. This is one of the biggest differences between the SSL version 3.0 and TLS version 1.0 specifications.

As mentioned, the record layer compresses and encrypts data. Because of the way this is implemented, when receiving packages the TLS layer needs to know that the packages arrive in the correct order to be able to successfully decrypt the data. This means that TLS requires an underlying transport protocol with such features, e.g. TCP.

1.3 The Handshake Protocol

The handshake protocol is responsible for negotiating cipher suites and in general setting up a connection between to parties. The handshake can be done in several different ways, depending on the requirements of the connection. Most often the server needs to be authenticated, but not necessarily.

1.3.1 Setting up Encrypted Communication

When a client wishes to establish a secure channel between itself and a TLS-capable server, the first thing it sends is the *ClientHello* message. Among other things, the *ClientHello* contains a list of the cipher suites the client supports. Upon receiving the *ClientHello*, the server will answer with the *ServerHello* message. In the *ServerHello* message, the server tells the client which cipher suite it has chosen for use with this connection. In TLS the server is required to select the first cipher suite it supports from the prioritised list offered by the client. There is however no such requirement in SSL. In other words, if the server is using SSL, it can—without violating the specification—choose the client's lowest ranking cipher suite.

The communicating parties are now ready for key exchange. The server initiates this by sending the *ServerKeyExchange* message. This message's contents depend on which cipher suite was chosen, but usually it contains the server's public key. The server then ends its part of the negotiation with the *ServerHelloDone* message.

The client will then generate a so-called *premaster secret*. The premaster secret is later used for creating the *master secret*, which again is used to derive various required values for symmetric encryption. This includes, but is not limited to, session keys and initial vectors. For generation of the master secret and subsequent secrets, the *pseudorandom function* (PRF) is used (see section 1.5). In the case of RSA key exchange, the client encrypts the premaster secret with the server's public key, and then sends it in the *ClientKeyExchange* message. Both server and client are now capable of deriving the session key. The client and server then exchange the finishing messages which signal the initiation of encrypted communications. These messages are the *ChangeCipherSpec* message which signals that the key negotiation is complete and the *Finished* message which contains a summary of what was just agreed. The involved parties can then verify that the negotiation was a success.

If the negotiating parties agreed on using Diffie-Hellman for key exchange, the *ClientKeyExchange* message will not contain an encrypted premaster secret, but rather the client's public Diffie-Hellman value needed by the server for calculating the same premaster secret as the client.

Figure 2 shows an overview of this handshake.

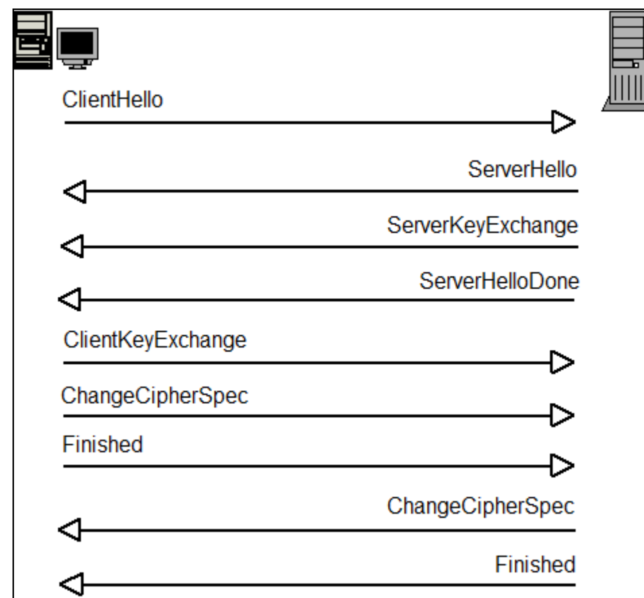


Figure 2: Handshake protocol (no authentication)

1.3.2 Server Authentication

Notice that in section 1.3.1, we never authenticated neither the client nor the server. All that was done was setting up an encrypted communication channel between a server which possessed a public and private key and a client. The communication was, in essence, anonymous and therefore wide open to a typical man-in-the-middle attack, where an adversary impersonates another entity.

To authenticate a server, we need to involve a trusted third party; enter the certificate. A certificate is a block of data which contains a server's public asymmetric key (at least in the case of RSA key exchange), and proves that a given server is the server it claims to be. The certificates are issued by a certificate authority, which has to be unconditionally trusted by the client. The validity of a site's certificate can be verified as the certificate is signed by a trusted certificate authority. This way one can verify signatures up the certificate chain to finally be able to verify that a site's certificate is valid.

For a client to be able to authenticate the server, we'll need to introduce one new TLS message, namely the *Certificate*, and we need to modify the contents of the *ClientKeyExchange* message. The *Certificate* is sent in place of the *ServerKeyExchange* message, as a certificate will both authenticate a server and contain the server's public key (alternatively fixed Diffie-Hellman public values). It is up to the client to verify that the certificate is valid and that the certificate belongs to the entity it is communicating with. By knowing the public key of the most common and trusted certificate authorities, the client can validate the signature on the certificate using asymmetric cryptography (typically this is done by using RSA).

In the *ClientKeyExchange* message the client encrypts the session key with the public key contained in the server's certificate. This way the server has to prove that it possesses the private key companion of the public key in the certificate to gain access to the session key. Only now is the server authenticated.

See Figure 3 for an overview of the handshake with server authentication.

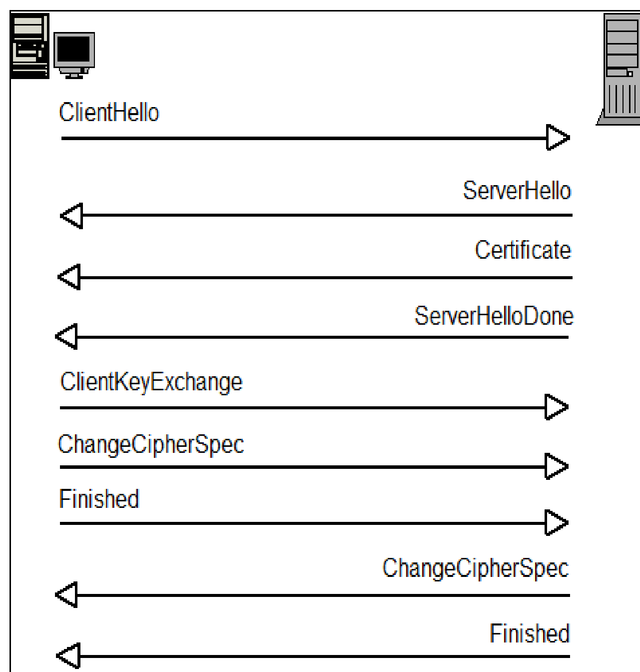


Figure 3: Server authentication.

1.3.3 Mutual Authentication

Due to the nature of the World Wide Web it is most important to authenticate the server, especially in cases with e-commerce involved. However, in some cases, a two-way authentication is desirable, or even required.

The protocol is basically the same as in section 1.3.2, with the exception that after the server has sent *Certificate*, it also sends a *CertificateRequest* message. This will prompt the client to provide a certificate of its own to prove its identity towards the server. The client then sends its own *Certificate* message. Note that even though the *Certificate* message sent by the server made the *ServerKeyExchange* message redundant, the client still needs to send *ClientKeyExchange*. The reason for this is that the *ServerKeyExchange* message contained the server's public key which was to be used for symmetric key exchange. The *ClientKeyExchange* message on the other hand contains the symmetric session key itself, and thus needs to be sent.

As with the server in section 1.3.2, the client too needs to prove that it possesses the private key companion to the public key in the certificate. This is done by sending the *CertificateVerify* message. It contains a summary of previous events, digitally signed with the private key. The server can then check if the events truly happened, and then verify the message by using the public key in the certificate. The client has now proven its identity towards the server.

Although it is rarely used, it is also possible to only authenticate the client. This goes to show the great flexibility of TLS.

See Figure 4 for an overview of a mutual authentication handshake.

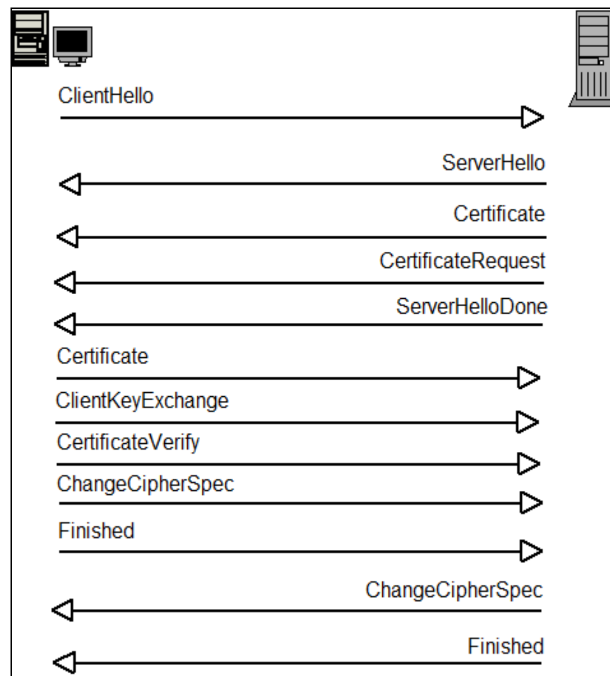


Figure 4: Client and server authentication.

1.3.4 Resuming a Session

A client and server may resume a previously negotiated session. To reduce the overhead of a full handshake which requires time expensive public key calculations, the specifications provide a sort of truncated handshake for resuming a session. The client sends along the session ID which was generated during the last session (or the session the client wishes to resume) in the *ClientHello* message. If the server accepts the resumption of the session, it replies with a *ServerHello* containing the same session ID. The handshake is concluded by exchanging *ChangeCipherSpec* and *Finished* messages.

If the server for some reason does not wish to resume the session, it can just generate a new session ID to pass along in the *ServerHello* message. A full handshake will then follow.

Figure 5 shows the truncated handshake for resuming a session.

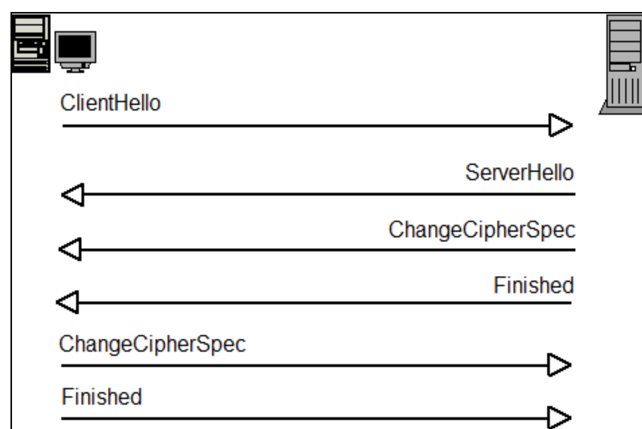


Figure 5: Handshake when resuming a connection.

1.4 Other Protocols in TLS

In addition to the handshake protocol, there are defined two more protocols in TLS, along with what are called the application protocols.

1.4.1 Change Cipher Spec Protocol

The *ChangeCipherSpec* message is used to change from one cipher strategy to another, most notably from no encryption to encryption. The change cipher spec protocol consists only of the *ChangeCipherSpec* message, which is a one-byte message, containing the value “1”. As one can see from section 1.3.1, the message can be seen as a part of the handshake, but because of its importance, it is defined as its own protocol.

1.4.2 Alert Protocol

The alert protocol is used for emergency messages which signal an error or other faulty condition to the other communicating party. The alert protocol consists of one specific type of message, which has two data fields: one field for indicating the *alert level* (severity level), and one which is an *alert description*. The alert level can be one of two values; *warning* (indicated by the number 1) or *fatal* (indicated by the number 2). A fatal alert will require the parties to immediately bring down the communication link, thus terminating the current TLS session and discarding any secrets associated with it (session identifier, keys, premaster and master secrets, etc). Upon receiving a warning alert, the parties may decide whether or not to terminate the session. The session may however not be used in future connections.

In the SSL specification there are defined 10 different alerts which may be used as alert description. The TLS version 1.0 specification adds a host of new alerts, divided into one *closure alert* and 22 *error alerts*. The TLS Extensions specification [RFC 3546] adds another 5 error alerts. In TLS, both of the communicating parties are required to send the closure alert (*close_notify*) upon the finalisation of the session so that both parties know that the session has been properly ended, thus avoiding truncation attacks where an attacker may prematurely manage to terminate the session.

1.4.3 Application Data Protocol

The application data passed down from the application layer is taken care of by the record layer. The application data is fragmented and compressed, then encrypted as per negotiated cipher specs the same way as TLS messages before being sent down to the transport layer.

1.5 Pseudorandom Function

The PRF is one of the most notable differences between SSL version 3 and TLS. Whereas SSL used a number of MD5 and SHA-1 iterations to generate key material from the premaster secret, the PRF in TLS uses repeated rounds of HMAC with both MD5 and SHA-1 as underlying message digest algorithm.

The objective of the PRF is to expand a relatively short input to a longer, cryptographically pseudorandom output. In TLS, this is used when expanding the premaster secret into a master secret, and then expanding the master secret to be able to derive different session keys, MAC secrets and initial vectors. The PRF is also used in the *Finished* message of the handshake protocol.

The PRF is defined as

PRF(secret, label, seed) = P_MD5(S1, label||seed) XOR P_SHA-1(S2, label||seed)

where *secret* is the secret which are to be expanded (e.g., the premaster secret), *label* is an identifying text string and *seed* is used to as a salt to the function (e.g., nonces sent between client and server in the *ClientHello* and *ServerHello* messages). P_MD5 and P_SHA-1 indicates repeated, concatenated HMAC rounds of the specified algorithm and is the expansion functions of the PRF. *S1* and *S2* are each one half of the *secret*.

1.6 Current Cipher Suites in TLS

1.6.1 General Cipher Suites

The TLS specification opens for TLS to support a large number of symmetric encryption algorithms and key sizes, along with various algorithms for key exchange and MAC generation.

In TLS version 1.0 the supported symmetric ciphers as defined by [RFC 2246] are: 3DES, DES, IDEA, RC4 and RC2, but due to the modular nature of the TLS specification's cipher suite definitions other ciphers like AES are in quite common use in current implementations. AES in TLS is described in [RFC 3268], and is seeing a wider and wider implementation in current applications supporting TLS and is currently considered to be the most secure symmetric encryption algorithm alongside 3DES. The MACs defined by [RFC 2246] are MD5 and SHA-1.

As long as a cipher suite does not require additional or modified handshake messages anyone is in theory free to add a cipher suite to TLS. This is done by issuing an RFC detailing the implementation of the suite, for example as done with AES.

In the TLS specification only one cipher suite is regarded as mandatory, namely the TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA suite. This suite uses ephemeral Diffie-Hellman with Diffie-Hellman parameters in a certificate signed by using DSS for key exchange and authentication. Further it uses three DES passes (Encrypt-Decrypt-Encrypt with three different keys) using the CBC operational mode for encrypting traffic and SHA-1 in the HMAC for MAC generation.

In the TLS version 1.1 specification which is still a work in progress, the mandatory cipher suite has been changed to TLS_RSA_WITH_3DES_EDE_CBC_SHA (which uses RSA for key exchange and authentication, and the same algorithms for encryption and MAC as the suite in the TLS version 1.0 specification).

1.6.2 Kerberos Cipher Suite

It is possible to use Kerberos [RFC 1510] as a cipher suite instead of the public key solution with certificates. Using Kerberos is often preferable in organisations that already have established authentication systems based on symmetric cryptography. With Kerberos in TLS it is possible to achieve mutual authentication and establishment of a master secret without using the traditional certificate approach.

The negotiation of using Kerberos is done through the client and server hello messages. The server's *certificate*, the client's *CertificateRequest* and the *ServerKeyExchange* is skipped. This is done because the master secret will be generated using the client's Kerberos credentials. The client must first obtain a service ticket for the TLS server from the Kerberos server. The pre-master secret is then encrypted under the Kerberos session key and sent to the TLS server along with the Kerberos credentials. Once the *ClientKeyExchange* message is received, the server's secret key is used to decrypt the credentials and extract the pre-master secret. Then the master key is derived from the premaster secret. Mutual client-server authentication is achieved, because the TLS server proves the knowledge of the master secret in the *finished* message.

The use of Kerberos in TLS is almost exactly as using an ordinary public key algorithm in TLS. No new messages are necessary in the TLS protocol.

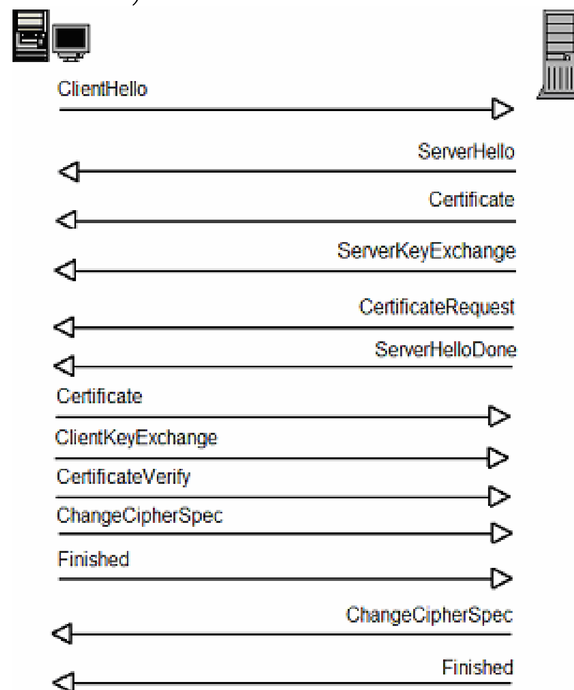
1.6.3 Cipher Suites and Perfect Forward Secrecy

In the attribute *perfect forward secrecy* (PFS) lies that if the long term secret (e.g., Diffie-Hellman public values) of a server is compromised, an attacker will not be able to decrypt previously made

encrypted communication. In other words, an attacker who knows a long term secret will not be able to reconstruct previous session keys if PFS is provided.

It is important to be aware that not all cipher suites in SSL and TLS provides PFS. When using RSA, the session key is encrypted with the server's public RSA key. Obviously, it follows that if the server's private RSA key is compromised, an attacker who has been monitoring and storing traffic will be able to decrypt all information needed for deriving the session key. It follows that all past communication is compromised.

To obtain PFS in TLS, a key exchange in addition to the certificate will have to be done. This way the peers will both be able to authenticate each other and provide either ephemeral RSA keys or ephemeral Diffie-Hellman parameters. It is then these keys or parameters which are used in either premaster secret exchange (in the ephemeral RSA case) or premaster secret calculation (in the ephemeral Diffie-Hellman case).



Figur 6 - Mutal authentication and PFS

The bottom line is that with careful selection of cipher suites, perfect forward secrecy is obtainable in SSL and TLS. The TLS version 1.1 draft [Dierks] recommends that users wanting PFS should use only cipher suites based on ephemeral Diffe-Hellman.

Note that while PFS makes sure past session keys are left uncompromised, future session keys are still compromised if the server's long term secret is leaked, in addition to that the connection is wide open for man-in-the-middle attacks.

2 TLS Extensions

[RFC 3546] describe several extensions to TLS version 1.0 that most likely are going to be implemented in the future. The main focus of [RFC 3546] is to extend functionality through the TLS protocol message formats. The main goals for TLS extensions is directed against minimising use of bandwidth in constrained access networks and conservation of the clients need for memory.

The specific proposed solutions in [RFC 3546] are:

- Allowing the clients to send the server name to the TLS server. This is practical for servers on a single network address that provide several "virtual" servers. See section 2.1.1.
- Making it possible to negotiate the maximum fragment length to be sent. Small fragment size is often required in environments with low memory clients and in networks with low bandwidth. See section 2.1.2.
- Conserving memory on the clients through use of certificate URLs instead of storing certificates on the client. See section 2.1.3.
- Make it possible to inform the server about which CA root keys each clients have. This to prevent handshake failures in communication with clients who store small amount of CA root keys because of limited memory. See section 2.1.4.
- Implement the possibility of using truncated MACs for integrity control. This to conserve bandwidth when sending MACs over the network. See section 2.1.5.
- Allow clients and servers to negotiate that the server can send client certificate status information during the TLS handshake. With client certificate status information it is possible to avoid sending a rather large Certificate Revocation List (CRL) over the network and therefore save bandwidth. See section 2.1.6.

When applying extensions to TLS it is important that the new versions are backwards compatible with current and previous versions of TLS version 1.0. In [RFC 3546] this problem is primary solved by implementing a new set of extended client hello messages. TLS version 1.0 servers will accept extended client hello messages, even if the server does not understand the extensions. [RFC 3546] also only propose extensions that don't need new mandatory responses from a TLS server. So if the server only supports TLS version 1.0 the client just won't get the extensions to work, in other words the server won't reply with an extended server hello message.

There are also many proposed internet drafts on extensions for the new TLS version 1.1 or other future versions. Beneath follows an introduction to some extensions that are proposed.

2.1 Extended TLS Handshake

All the TLS extensions suggested in [RFC 3546] affects the *ClientHello* and/or the *ServerHello* messages sent between the client and the server during the TLS connection phase. Beneath are brief descriptions of the new extensions.

The extended client hello message format contains an additional data field. This field contains a list of available extensions. It tells the particular extension type in one part of the field, and the extension data in another. When a server receives an extended client hello, there are three possibilities that could occur.

- The server supports TLS extensions and replies with an extended server hello.
- It may choose not to reply with the extended server hello message.
- The server doesn't support extensions, thus it will reply with the standard *ServerHello*.

The client may in the last two cases choose whether to proceed without extensions or abort the handshake.

Below we have described some of the proposed extensions and their uses.

2.1.1 Server Name Indication

Here the extension type will be *server_name*, and the extension data field will contain a *ServerNameList*. The server name would be the DNS hostname for the particular server. IPv4/v6 addresses are not permitted. DNS is by now the only supported server/host name, but it's

possible for other name types to be added later. When a server receives an extended client hello with extension type *server_name* and returns with an extended server hello, the server extension type shall be *server_name*, and the extension data field shall be empty.

This extension is intended to allow the client to tell the server which server it is contacting. This may be desirable for the clients to facilitate secure connections to servers that host multiple 'virtual' servers at a single underlying network address.

[Banes] describe an update to this extension, where one can use an email address in place of the server name to enhance the usability of this extension, however this is currently only a draft and considered work-in-progress.

2.1.2 Maximum Fragment Length Negotiation

This extension specifies a way to give plaintext fragments a predefined fixed length, different from the original size of 2^{14} bits. The extension type in the extended client hello will be *max_fragment_length*. The extension data field shall contain the desired fragment length. The allowed values for this field are 2^9, 2^{10}, 2^{11}, and 2^{12} bits. Servers that receive an extended client hello containing a *max_fragment_length* extension, returns an extended server hello with extension type *max_fragment_length*, where the extension data shall contain the same fragment length as the client requested. If the server does not support this extension, or does not allow usage of this, it returns a regular server hello, and if the server receives an extended client hello with an illegal desired fragment length, it must abort the handshake with an *illegal_parameter* alert. This applies the client as well.

Once the maximum fragment length has been successfully negotiated, they immediately start to send fragments with the negotiated length. This means that even the handshake messages will be sent with the new fragment length. This extension is suggested as an option for clients with memory limitations and bandwidth limitations.

2.1.3 Client Certificate URLs

This extension will make it possible for clients to authenticate themselves to the server without possessing their certificates on their own computer. The way to obtain this is to allow clients to pass URLs to where the certificates are located instead of keeping the certificates local and passing them during the TLS handshake as defined in TLS version 1.0. The main goal for this extension is to save memory on the clients. If this extension is desirable, clients sends an extended hello message with extension type *client_certificate_url*. The extension data field in this case shall be empty. The server may then indicate that it is willing to accept certificate URLs by include the *client_certificate_url* extension with an empty *extension_data* field in the extended server hello message.

When the extended hellos are successfully completed, the client may send certificate URLs in place of a certificate. This way of handling authentication introduces a whole new aspect of security. To do the authentication the server may now contact other servers on other URLs to authenticate the clients trying to set up a TLS connection. Even though it may be extremely easy for an attacker to gain access to a client's certificate, the client must still prove the ownership of the certificate by knowing the private key companion to the public key which is stored in the certificate.

Trust, correct servers and correct URLs are keywords that must be well thought through before implementing and using such an extension.

2.1.4 Trusted CA Indication

A client that only contains a small amount of CA root certificates, often due to a small memory limitation, may want to tell the server which CA he possesses to avoid repeated handshake failures. To do this, the client may include an extension called *trusted_ca_keys* in the client hello. The *extension_data* field will then contain *TrustedAuthorities*. The *TrustedAuthorities* data is a list of CA root certificates that the client possesses. When a server receives the client hello message containing this *trusted_ca_keys* extension, it may use this information when choosing an appropriate certification chain to return to the client. The server will then include a *trusted_ca_keys* extension in the extended server hello message. Here the *extension_data* field shall be empty.

2.1.5 Truncated HMAC

The TLS cipher suite uses either an MD5 or a SHA-1 based HMAC to authenticate record layer communications. The entire output from the HMAC is used as the MAC tag, but it could be desirable to save bandwidth by truncating the HMAC output to 80 bits when the MAC tag is created. For the negotiation of 80 bits truncated HMAC, the client may include a *truncated_hmac* extension in the client hello message. When a server receives this message, he may agree to use a truncated HMAC, and return a *truncated_hmac* extension in the server hello message. This extension will not have any effect if there are added new cipher suites that don't use HMAC and the session negotiates one of these suites. If there has been negotiated a HMAC truncation, then this is passed to the TLS record layer along with the other negotiated security parameters. Then the client and server must use truncated HMACs during the session. This means that the *CipherSpec.hash_size* is set to 80 bits and only the first 80 bits of the HMAC output are transmitted and checked. This extension does however not affect the calculation made by the PRF as part of handshaking or key derivation. Using this extension will most likely not affect the security aspects in the extended TLS protocol.

2.1.6 Certificate Status Request

In order to avoid transmissions of CRLs and save bandwidth, clients may want to use a certificate-status protocol (like Online Certificate Status Protocol (OCSP) [RFC 2560]) to instantly check the validity of server certificates when needed. By using this extension, this information can be sent in the TLS handshake, and therefore save recourses. The clients then may include an extension called *status_request* in the client hello message to indicate their desire to receive certificate status. Then the *extension_data* field should include a *CertificateStatusRequest*. The *CertificateStatusRequest* will include a list of OCSP responders that the client trusts. If the server receives a client hello message including this extension, it may then return a suitable certificate status response to the client along with their certificate. The response is returned with the certificate by sending a *CertificateStatus* message. If this message is sent, it must contain a *status_request* extension, where the *data_field* will contain an *OCSP_response* which includes a complete OCSP response. Only one OCSP response may be sent back to the client.

2.2 Future Cipher Suites Based on Extensions

There have been proposed several new cipher suites for TLS which are based on TLS extensions, that is, they make use of the extended handshake. None of the following cipher suites are more than work-in-progress, but they are presented here as examples of what kinds of cipher suites the future will bring TLS.

2.2.1 OpenPGP Keys for TLS Authentication

There is an interesting proposed internet-draft on applying OpenPGP keys for TLS authentication by [Mavroyanopoulos]. The main goal is of course to implement the use of a new trust model to TLS, more specific the "web of trust" model of PGP. As has been described in earlier sections, TLS uses standard certificate based X.509 PKI framework for authentication. An OpenPGP implementation will be a new way of authenticating with the well used and tested

PGP approach. To use OpenPGP the hello messages must include some information. [Mavroyanopoulos] introduces a new extension named *cert_type*. The *cert_type* value in the client and server hello messages must include OpenPGP as an alternative. If only the X.509 certificates are supported by the server the field may be omitted or the server may terminate the connection with an *unsupported_certificate* alert message. If the OpenPGP certificate type is agreed upon by the client and server an OpenPGP key must be included in the certificate message. The OpenPGP key must contain a public key for the selected key exchange algorithm, RSA public key for encrypting and DSS-signed Diffie-Hellman public parameters or RSA key for signing. The internet draft also propose sending the OpenPGP fingerprint in the certificate, instead of sending the entire OpenPGP key, this to conserve bandwidth and memory requirements. The implementation of OpenPGP in TLS should be straightforward and introduce the "web of trust" model into TLS.

2.2.2 Elliptic Curve Cryptography for TLS

Using ECC for public-key crypto system is attractive, especially in environments with low bandwidth and memory constrained clients, because ECC offers high security with relative small key size compared to today's asymmetric crypto systems. Based on [Gupta] and [Lenstra], Table 1 shows a comparison of key size, based on the best-known algorithms for attacking cryptographic algorithms. Smaller key sizes result in power, bandwidth and memory conservation.

Symmetric	DH/DSA/RSA	Elliptic Curve (ECC)
80	1024	163
112	2048	233
128	3072	283
192	7680	409
256	15360	571

Table 1 - Keysize comparison (in bits)

The Internet-Draft "ECC Cipher Suites for TLS" [Gupta] propose how to use elliptic curve cryptography in TLS. More specific the use of the Elliptic Curve Diffie-Hellman key agreement scheme and the use of Elliptic Curve Diffie-Hellman certificates and Elliptic Curve DSA for authentication. The use of elliptic curve cryptography may the future of secure communication.

To implement ECC in TLS, [Gupta] proposes the introduction of two new TLS extensions, namely the *Supported Elliptic Curve Extension* and the *Supported Point Formats Extension*. The extensions enumerate which curves the application supports and which point formats for point compression it supports. Encrypting a message with ECC may easily increase the message length by a factor of four [Stinson]. Therefore a trick called point compression is used to reduce the storage requirement for the points on an elliptic curve.

[Gupta] proposes to use ECC in conjunction with Diffie-Hellman, both for key exchange and certificate signing. The Diffie-Hellman and ElGamal cryptosystems are the two systems which are best suited for ECC application, and the Diffie-Hellman algorithm is pretty straight-forward (at least as far as that term can be used with regards to ECC) to convert from ordinary modular arithmetic to ECC calculations. All modular multiplication can be converted to addition of points on an elliptic curve. Similarly, all modular exponential calculations can be converted to multiplications of points on an elliptic curve.

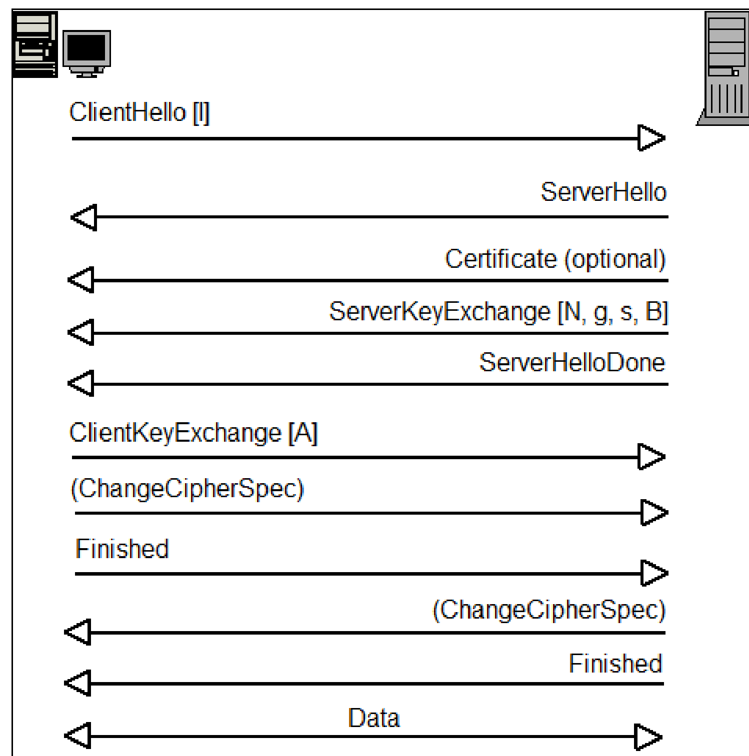
The draft proposes a set of key exchange and authentication algorithms, all based on Elliptic Curve Diffie-Hellman and various certificate signing algorithms.

2.2.3 Secure Remote Passwords for TLS Authentication

Today TLS uses PKI or Kerberos for authentication. These authentication methods are not well suited for the applications now being adapted to use TLS (like IMAP or FTP). These protocols are designed to use usernames and passwords for authentication. Being able to use usernames and passwords to authenticate the TLS will provide more security to the TLS connection than implementing a PKI in certain situations.

SRP (Secure Remote Password) [RFC 2945] allows the user to type a username and password without having to worry about eavesdropping [Taylor]. It uses a shared secret that can be used to generate encryption keys.

The SRP protocol can be implemented using the standard TLS handshake protocol, however with some modifications:



Figur 7: The SRP Protocol

Variable names:

N, *g*	group parameters (prime and generator)
s	salt
B, *b*	server's public and private values
A, *a*	client's public and private values
I	user name

The mechanism for re-using sessions for new connections and key renegotiations for existing connections will still work with the SRP protocol. When a client wants to re-use a session under SRP, it includes the SRP extensions that carry the username in the *client_hello* message. It requires a full handshake sequence in case the server can't or won't allow re-use of the session.

The protocol uses a unique verifier for the handshake, which is based on a salt (*s*), a username (*I*), the client's password and the group parameters (*N* and *g*). The algorithm makes use of a SHA-1 hash algorithm.

The verification of the user's password is done without sending the password in any form to the server. The password and the other variables which have been transferred between client and server are used when both the server and the client each generate a premaster secret. That way, when we get to the *Finished* message, it will not decrypt properly if the premaster secrets of the server and client differ. If they differ, the client has provided a wrong password, and no session will be set up.

2.2.4 Shared Keys in TLS

"Use of Shared Keys in the TLS Protocol" is an intenet-draft by [Gutman] that proposes use of symmetric keys instead of using CPU-intensive public-key algorithms. The use of symmetric keys that are shared in advance also have the benefit that it provide cryptographic authentication of both server and client without the use of certificates. This approach will simplify the TLS handshake protocol. In environments with little CPU power i.e. mobile devices the use of today's public-key-based handshakes takes a lot of recourses and time. If the hosts in the environments already have pre-shared symmetric keys it would be preferable to use these keys instead of computing new ones in the TLS handshake. Pre-shared keys may also be more convenient from a key management point of view. E.g. in networks where the connections is configured manually in advanced it may be easier with shared keys than the use of certificates or in another case where the communicating parties already have a mechanism for setting up a shared secret key.

With a pre-shared secret the TLS master secret can be derived from the shared key instead of deriving it through the handshake protocol. The idea is to make use of the handshake resume instead of the full handshake with public key exchange. The shared symmetric key is just seeded into the TLS session cache. When the client connects, the session resume takes over and the client and server "resume" the "phantom" session by seeding the cache.

The TLS master key is 48 byte, this is to big to be covered by a single shared symmetric key. To use the shorter symmetric keys, the TLS pseudorandom function must be used to produce the master key which is seeded into the session cache. When the PRF function is used with the shared symmetric key the result is a pre-master secret. Then nothing else is needed to be changed or implemented for using pre-shared keys in TLS. The use of a pre-shared key also opens up for using passwords or pass phrases instead of symmetric/public keys. This is of course less secure then symmetric or public keys, but it could be preferable in some environments. See section 2.2.3.

"Pre-Shared Key Ciphersuites for TLS" is another Internet draft by [Eronen] and H. Tschofenig published February 6, 2004. The draft specifies new cipher suites to be used in a pre-shared symmetric key TLS setting. Here the pre-master secret is build as follows: concatenate 24 zero byte, a SHA-1 hash of the pre-shared key (20 byte) and at last 4 zero byte. In other words only the HMAC-SHA1 part of the TLS pseudorandom function is used. Since the pre-shared secret key can be of variable length depending on algorithm used the SHA-1 hash is used to always end up in a 48 byte pre-master secret. The proposed symmetric chipersuites are as follows:

```
TLS_PSK_WITH_RC4_128_SHA
TLS_PSK_WITH_3DES_EDE_CBC_SHA
TLS_PSTK_WITH_AES_128_CBC_SHA
TLS_PSTK_WITH_AES_256_CBC_SHA
```

In this draft the use of the pre-shared keys is implemented through a slightly modified handshake protocol. The handshake just doesn't include the values: *certificate*, *certificaterequest* and *certificateverify* in the client/server packet exchange since they are no longer necessary. Since it's likely that the both the server and client may have pre-shared keys with several different parties, the client

indicates which key to use by including a "pre shared key identity" in the *ClientKeyExhange* message. To help the client in selecting which identity to use, the server can provide a "pre-shared key identity hint" in the *ServerKeyExhange* message. The format of the identity and identity hint can be as easy as a user name or a host name.

Both the "Pre-Shared Key Ciphersuites for TLS" and the "Use of Shared Keys in the TLS Protocol" draft introduce a new important extension to TLS. With pre-shared keys in TLS the usability of TLS extends widely and it become possible to use the protocol other environments and settings than to days TLS version where only public key algorithms is supported in the handshake.

3 TLS' Future

3.1 TLS version 1.1

Version 1.1 of the TLS specification [Dierks] is mostly an attempt to clarify some things from the previous version and fix some minor flaws, but there are a couple of things which are of importance.

- It acknowledges the existence of TLS extensions as defined in [RFC 3546].
- It informs implementers that TLS' authenticate-then-encrypt approach may be open for chosen plaintext attacks when using some cipher suites.
- It fixes vulnerabilities against different attacks on the CBC operational mode.
- It changes the mandatory cipher suite's key exchange and authentication algorithm from ephemeral Diffie-Hellman to RSA.
- It recommends that 40 bit cipher suites should *not* be enabled by default in applications supporting TLS.

3.2 The Future of TLS

SSL was first implemented into Web Wide Web servers and browsers to secure communications between a server and clients. Today, HTTP traffic between a Web server and clients are still the main application of TLS. When speaking strictly amount of sites, online banking services and online shopping malls are most common. When operating at the maximum capacity of a one gigabit connection, a typical Web shop may have to do as many as 10 to 20,000 TLS handshakes per second, depending on the amount of data being returned on each request [Reynolds]. The process of going through such an amount of TLS handshakes is immensely expensive in terms of CPU time due to complex RSA calculations and master key generation. Not all servers are up to this task, which may result in extremely slow response times which again may result in lost revenues for the Web shop in question. The solution to this is of course doing the heavy mathematical calculations in hardware. Such hardware has existed for quite some time, and may work very well in most cases today. However, the need for TLS handshake processing will only increase with more applications of TLS, including other applications of TLS on the Web. When this happens, our current solutions may not cut it. As the thought of offloading the server's main CPU with other dedicated processors is good, we might put more of the load on these secondary processors. That means doing more of the handshake processing off the main processor. This will then include all mathematical RSA calculations as today, but also more of the master secret generation based off of the premaster secret and TLS handshake message creation (including record layer wrapping).

In the near future we believe it is very likely that TLS will enjoy more widespread acceptance in other applications than Web browsers. We have seen lately that the protection of personal information is increasingly important to the average user of Internet applications where confidentiality and integrity up until now has been totally ignored. The increased public

awareness of privacy protection and an aversion towards an even remotely Orwellian type of big brother society, we believe will dramatically increase the need for secure communications and encryption on the Internet. Some real life examples of such new applications of TLS are better securing of the authentication process of SMTP and POP servers, and securing of traffic on Internet Relay Chat networks and in peer to peer Instant Message protocols like MSN, ICQ and AIM. There should not be any major hurdles with regards to implementing TLS into applications which uses these protocols even today.

Not only our personal computers are connected to the Internet. In a world where more and more of our everyday appliances are interconnected via the Internet, the need for confidential communication will dramatically increase. In such devices of limited hardware capabilities, TLS extensions which focus on decreased bandwidth usage will come to their right, in addition to special hardware capable of doing most of the logics involved in TLS handshake processing.

It is easy to imagine the now almost clichéd refrigerator which connects to the local grocery store to order new milk bottles using a secure TLS connection to communicate the grocery store. After all—the amount of information one can infer from monitoring the refrigerator's communication with the grocery can be quite a violation of a household's privacy.

3.3 Visions

In this section we will give some suggestions on extensions and other improvements on the TLS protocol.

3.3.1 Biometric Authentication Extension

The intension of this suggestion is to improve authentication, and limit the use of PKI, certificates or passwords. We imagine using the TLS handshake as suggested in [RFC 3546], and use the *extension_data_field* in the *extended_client_hello/extended_server_hello* messages to perform the biometric authentication. The authentication procedure would be something like this:

The client sends an *extended_client_hello* message including a live image captured with a standard web camera containing the user's face. This is *image1*. The client stores *image1* temporary. The client also sends a list containing which ciphers he prefers to use. Together with *image1*, he sends a hash of it to verify its authenticity. The server then creates a hash of the received image1 and compares it to the hash value that was sent. If they are equal it stores *image1* temporary. Comparing the hashes is mainly to detect transfer errors. Then the server creates a challenge and sends it to the client along with the first cipher selection from the clients cipher list it supports in the *extended_server_hello* message. The challenge would be a random (possibly one-time) generated string that the user is asked to write down on a piece of paper in a given format, and hold it up in front of the web camera in a way that makes the web camera capture both the challenge and the user's face in one shot. This is *image2*. The user has a limited period of time (like 10 seconds) to answer this challenge. *Image2* is then sent to the server along with a hash of it in the *certificate* message in stead of a standard certificate.

The server then creates a hash of *image2,* and compares the hash values. The server then performs a biometric verification on the facial images it has received from the user, comparing *image1* with *image2*. Then it compares the response from the user with the challenge. If both the facial images are equal and the challenge/response is equal, the user is authenticated. It then creates a key derived from *image1* and *image2*. This is the session key. The client does the same, and both the client and the server now knows the session key. This key is used to generate the key stream for the session. We achieve perfect forward security if the challenge is unique and the image sent from the client really was captured with the web camera and not a previously captured and used image. The secured connection is now ready.

This method only proves that the client remains the same during the handshake. To verify who the user is, a local biometric description of the clients must exist on the server.
This protocol is also vulnerable for man-in-the-middle (MitM) attacks. To avoid this some sort of pre-distributed secret might be used. Another solution to avoid MitM attacks might be to establish a TTP who possesses faces of all TLS biometric extension users that might want to set up a connection to the server. When setting up a connection, the server will contact the TTP to receive the private key for the user authenticated from the received picture. Now only the server and the person on the picture (client) would know the private key or the secret. If the picture was of a MitM adversary, the server would look up his private key and not the correct key that the real client would posses. It is crucial at this point that the server authenticates to the TTP in some secure way. The protocol then continues very similar to what described above, but the session key will now be derived from both *image1* and *image2*, together with the private key. This will eliminate MitM attacks between the client and the server.

The Biometric Authentication Extension suggestion is hardly any real extension or suggestion of one, but rather a thought of technology which might be used in the future of TLS.

4 References

[Banes]	J. Banes, C. Crall, "*Update to Transport Layer Security (TLS) Extensions*", 2003. http://www.ietf.org/internet-drafts/draft-ietf-tls-emailaddr-00.txt
[Dierks]	T. Dierks, E. Rescorla, *"The TLS Protocol Version 1.1"*, 2004. http://www.ietf.org/internet-drafts/draft-ietf-tls-rfc2246-bis-06.txt
[Eronen]	P.Eronen, H. Tschofenig, "*Pre-Shared Key Ciphersuites for TLS*", 2004. http://www.ietf.org/internet-drafts/draft-eronen-tls-psk-00.txt
[FIPS PUB 180-2]	NIST, "*Secure Hash Standard*", 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
[Gupta]	V. Gupta, S. Blake-Wilson, B. Moeller, C. Hawk and N. Bolyard, "*ECC Cipher Suites for TLS*", 2004. http://www.ietf.org/internet-drafts/draft-ietf-tls-ecc-05.txt
[Gutman]	P.Gutman, "*Use of Shared Keys in the TLS Protocol*", 2003 http://www.ietf.org/internet-drafts/draft-ietf-tls-sharedkeys-02.txt
[Hollenbeck]	Scott Hollenbeck, "*Transport Layer Security Protocol Compression Methods*", 2003. http://www.ietf.org/internet-drafts/draft-ietf-tls-compression-07.txt
[Lenstra]	A. Lenstra, and E. Verheul, "*Selecting Cryptographic Key Sizes*", Journal of Cryptology 14, p. 255-293, Springer-Verlag New York, LLC, 2001.
[Mavroyanopoulos]	N. Mavroyanopoulos, "*Using OpenPGP keys for TLS authentication*", 2004. http://www.ietf.org/internet-drafts/draft-ietf-tls-openpgp-keys-05.txt
[Reynolds]	E. Reynolds, "*The Increasing Demand for SSL/TLS Processing*", 2003. http://www.layern.com/SSLWP0001SSL02.pdf
[RFC 2104]	H. Krawczyk, M. Bellare, R. Canetti, "*HMAC: Keyed-Hashing for Message Authentication*", 1997. http://www.ietf.org/rfc/rfc2104.txt
[RFC 2246]	T. Dierks, C. Allen, "*The TLS Protocol Version 1.0*", 1999. http://www.ietf.org/rfc/rfc2246.txt
[RFC 2537]	D. Eastlake, "*RSA/MD5 KEYs and SIGs in the Domain Name System (DNS)*", 1999. http://www.ietf.org/rfc/rfc2537.txt
[RFC 2560]	M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams, "*X.509 Internet Public Key Infrastructure – Online Certificate Status Protocol – OCSP*", 1999. http://www.ietf.org/rfc/rfc2560.txt
[RFC 2945]	T. Wu, "*The SRP Authentication and Key Exchange System*", 2000. http://www.ietf.org/rfc/rfc2945.txt
[RFC 3268]	P. Chown, "*Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)*", 2002. http://www.ietf.org/rfc/rfc3268.txt
[RFC 3546]	S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, T. Wright, "*Transport Layer Security (TLS) Extensions*", 2003. http://www.ietf.org/rfc/rfc3546.txt
[Stinson]	D. Stinson, "*Cryptography – Theory and Practice*", Chapman & Hall / CRC, 2002.
[Taylor]	D. Taylor, T. Wu, N. Mavroyanopoulos, T. Perrin, "*Using SRP for TLS Authentication*", 2004. http://www.ietf.org/internet-drafts/draft-ietf-tls-srp-06.txt