CS-203 Network and Distributed System Security Project

Implementation of a ProxyInfo TLS Extension for enhancing security in Multi-proxy architecture

By

Karthik Ragunath Balasundaram **78806170**
Mihir Kulkarni **89485131**

Donald Bren School of Information and Computer Science
University of California, Irvine

Under the guidance of:

Dr. Gene Tsudik
Professor, Department of Computer Science
University of California, Irvine

Dr. Kasper Rasmussen
Post-Doc, Department of Computer Science
University of California, Irvine

Contents

Chapter 1

Introduction

In this project we propose to implement an extension for the Transport Layer Security (TLS) described as follows. Transport Layer Security (TLS) is a cryptographic protocol that provides communication security over the Internet. TLS encrypts the segments of network connections above the Transport Layer, using asymmetric cryptography for key exchange, symmetric encryption for privacy, and message authentication codes for message integrity. TLS is often used to protect HTTP traffic from eavesdropping and tampering. In many commercial scenarios a HTTP proxy is used, for instance, to allow caching, to provide anonymity to a client, or to provide security by using an application-layer firewall to inspect HTTP traffic on behalf of the client.

A single TLS session cannot be used to protect traffic between a client and a server when a proxy is present, as the proxy needs access to unencrypted traffic to apply its policies. It is possible to have separate TLS sessions between the client and the proxy, on one side, and the proxy and the server on the other side; however the presence of the proxy removes the client's knowledge about the server. In addition the client cannot determine the security level of the TLS session between the proxy and the server. Without this knowledge, the client has no way to decide whether to trust the server, or not. This is especially problematic when the client trusts multiple different servers for different applications, or trusts servers from different domains. Hence we have implemented a TLS extension that will allow a client to establish a connection to a server, through any number of proxies, and have enough information about the security of each individual link, to make an informed decision about the trustworthiness of that server. In short, the client will also have the guarantee of a secure connection not only between itself and the proxy but also for the links from the proxy to the server. This will help in improving the overall security of the communication between the client and the server. We have defined a structure with certain vital information that would be exposed through extension APIs to the TLS protocol of the HTTP traffic which would help in concluding about the trustworthiness of every proxy that would be communicated back to the client before establishing a secure HTTP session with the server.

Chapter 2

Transport Layer Security

Transport Layer Security or TLS provides an entirely new protocol layer between the transport layer and the application layer, dedicated for security. In addition to this, 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 visualized in two layers. The record layer protocol is the secure communications provider. It acts as a bottom layer which comprises all of the other protocols, of which the Handshake protocol can be considered to be the most important one.

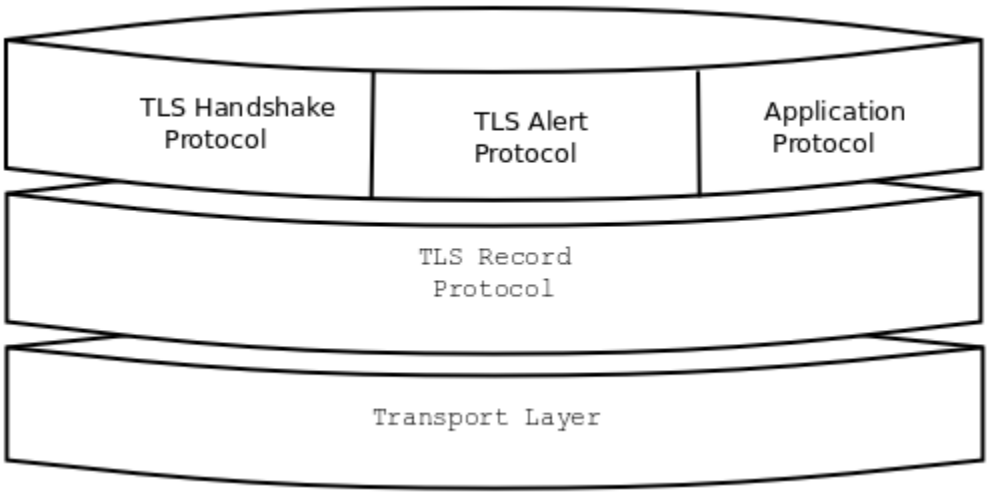

Figure 1.1: TLS Protocol Layers

The record protocol offers symmetric encryption, data authenticity, and optionally compression. The alert protocol offers some signaling to the other protocols. It can help informing the peer for the cause of failures and other error conditions. The alert protocol is above

the record protocol. The handshake protocol is responsible for the security parameters' negotiation, the initial key exchange and authentication.

2.1 The Record Layer Protocol

The Transport Layer Security makes use of the record layer protocol to encapsulate all data. It achieves this by accepting all of the different messages the other TLS protocols like the handshake protocol, change cipher spec protocol, etc. send. After this it wraps these messages 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 the content type which is the record layer protocol type contained in the record, which version of SSL or TLS is used (versions 3.0 and lower are SSL, version 3.1 is TLS) and also the length of the message. A message authentication code computed over the protocol message, with additional key material included can be computed and attached.

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.

2.2 The Handshake Protocol

As mentioned earlier, the handshake protocol is responsible for negotiating cipher suites and in general setting up a connection between two 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.

2.2.1 Setting up Encrypted Communication

When a client wishes to establish a secure channel between itself and a TLS-capable server, it first sends the ClientHello message. The ClientHello contains a list of the cipher suites the client supports. Upon receiving the ClientHello, the server will reply 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.

The communicating parties are now exchange their keys. 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 by sending 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, among others, session keys and initial vectors. For generation of the master secret and subsequent secrets, the pseudorandom function (PRF) is used. In the case when the RSA key exchange method is used, the client encrypts the premaster secret with the server's public key, and then sends it in the ClientKeyExchange message. Both server and client can now derive the session key which can be used amongst them for this particular session. The client and server then exchange the finishing messages which indicate 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. It is also possible for the involved parties to verify that the negotiation was a success.

For example, if the client and server 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 which is needed by the server for calculating the same premaster secret as the client.

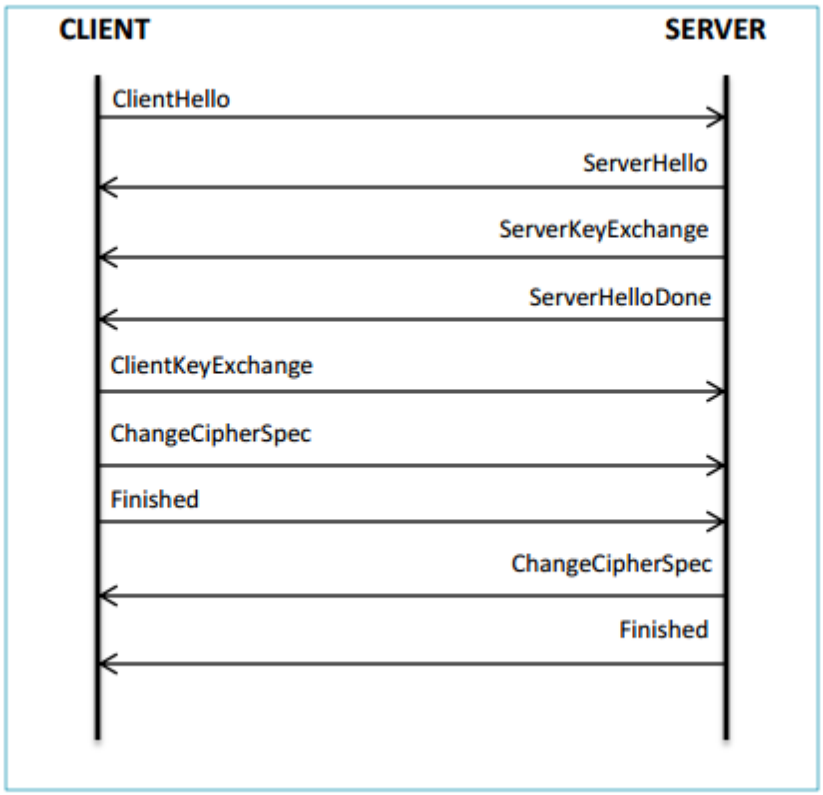

Figure 2.1: Handshake protocol (no authentication)

2.2.2 Server Authentication

In the above section, we never authenticated neither the client nor the server. What we ended up doing was setting up an encrypted communication channel between a server which possessed a public and private key and a client. The communication was anonymous and therefore clearly vulnerable 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 nothing but 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 trusted CA or 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. By doing this one can verify signatures up the certificate chain to finally be able to verify the validity of a site's certificate.

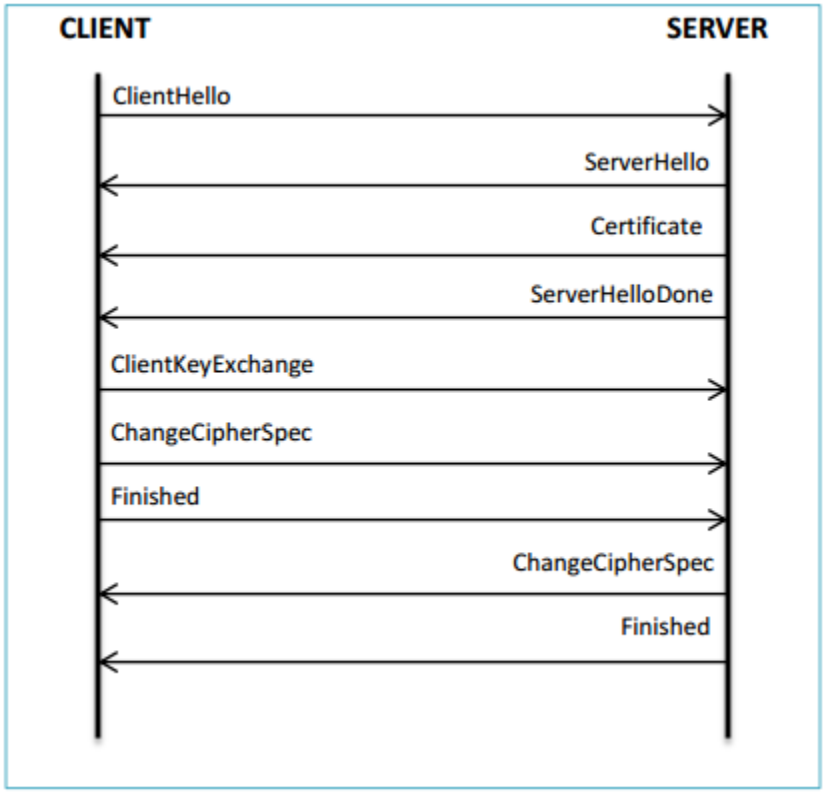

Figure 2.2: Server Authentication

For a client to be able to authenticate the server, we will need to add one more TLS message, which is called 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 whether 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 (most commonly 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. Now we can say that the server is authenticated.

2.2.3 Mutual Authentication

Due to the present 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 might be desirable, or even required. This protocol is basically the same as in section 2.2.2, with the exception that after the server has sent Certificate, it also sends a CertificateRequest message. This will force the client to provide a certificate of its own to prove its identity towards the server. The client then sends its own Certificate message. An important thing to note here is 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. On the other hand the ClientKeyExchange message contains the symmetric session key itself, and thus needs to be sent.

As with the server in section 2.2.2, 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 not used commonly, it is also possible to only authenticate the client. This shows the great flexibility of TLS protocol.

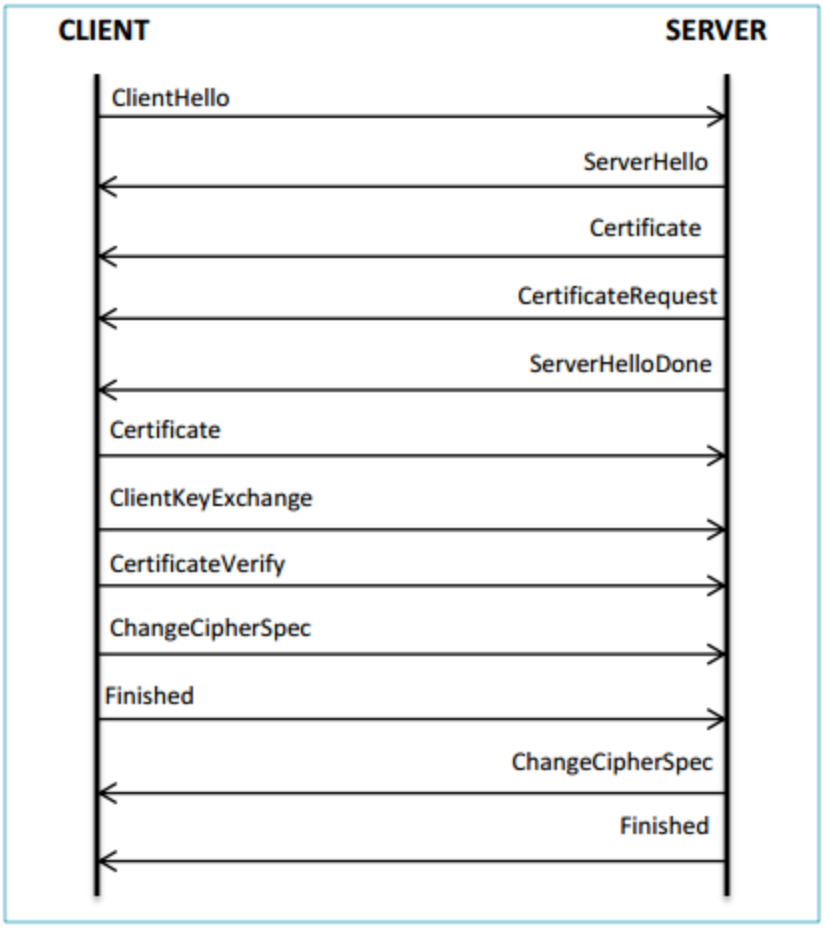

Figure 2.3: Client and server authentication

Chapter 3

GNUTLS

Gnutls is a library which offers an API to access secure communication protocols. These protocols provide privacy over insecure lines, and were designed to prevent eavesdropping, tampering or message forgery.It is a portable ANSI C based library which implements the protocols ranging from SSL 3.0 to TLS 1.2, accompanied with the required framework for authentication and public key infrastructure. Important features of the GnuTLS library include:

- Support for TLS 1.2, TLS 1.1, TLS 1.0 and SSL 3.0 protocols.
- Support for Datagram TLS 1.0.
- Support for handling and verification of X.509 and OpenPGP certificates.
- Support for password authentication using TLS-SRP.
- Support for keyed authentication using TLS-PSK.
- Support for PKCS #11 tokens and smart-cards.

The GnuTLS library consists of three independent parts, namely the “TLS protocol part”, the “Certificate part”, and the “Cryptographic back-end” part. The “TLS protocol part” is the actual protocol implementation, and is entirely implemented within the GnuTLS library. The “Certificate part” consists of the certificate parsing, and verification functions and it uses functionality from the libtasn1 library. The “Cryptographic back-end” is provided by the nettle library.

The GnuTLS library is licensed under the GNU Lesser General Public License; included applications are licensed under the GNU General Public License.

Chapter 4

TLS Extensions

[RFC 3546] describe several extensions to the TLS protocol 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 minimizing use of bandwidth in constrained access networks and conservation of the clients need for memory.

As of now there are various extensions available for use in the Gnutls library. Some of the supported extensions are:

- Maximum fragment length negotiation
- Server name indication
- Session tickets
- Safe Renegotiation

Our project implements another extension to the Gnutls library. The main use of this is to help the client application to get more information about the links that lie ahead of the first proxy that it is connected to. The details of this “proxy info” extension are in the following section.

Chapter 5

Design of ProxyInfo Extension

In its current functionality, the client cannot get information about the links which lie ahead of the proxy to which it is connected. The proxy may directly connect to the server or may connect to other proxies on a link which is not very secure and doesn't support the more secure cryptographic algorithms. But the client has no way of learning how secure the links are.

We have made available an extension to the Gnutls library which helps the client to get to know information about all the links which are used when connecting to the server through single or multiple proxies. This will in turn help the client application to decide whether the links are sufficiently secure to move ahead with the communication. Also, the client application can compare various different routes to the server from the security point of view and then make the final decision of choosing the most secure route.

An important thing to note is the fact that this additional information is made available to the client in the handshake phase itself and the client application need not exchange any extra packets over the network to get this information. The ProxyInfo data structure consists of:

- Cipher Algorithm
- Key Exchange Algorithm
- MAC Algorithm
- IP Address
- MAC Address

This data is made available for every link that gets established when connecting from client to server. Thus at the end of the handshake phase, the client application has the knowledge of the above protocols used in all the links. Based on this and the priority that it has for various different protocols the client can take the decision to either continue with the communication or break it.

An illustration of the working of this extension is shown below. Here there is a client, server and there are two intermediate proxies.

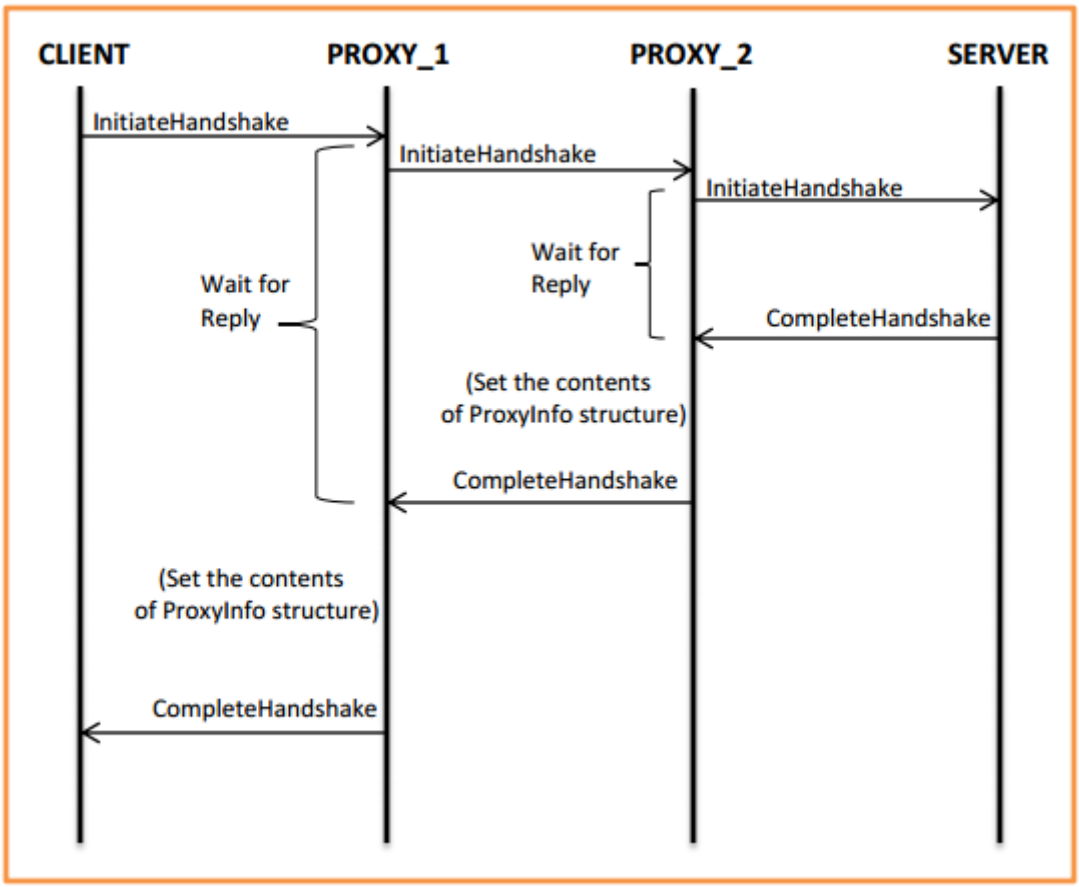

Figure 5.1: Working of ProxyInfo extension with two intermediate proxies

The client first starts the handshake with the first proxy. The first proxy before completing the handshake with the client, starts another handshake with the second proxy i.e. proxy_2. This second proxy, proxy_2 in turn pauses this handshake with the proxy_1 and starts another handshake with the server. Now, the server completes the handshake. The second proxy, proxy_2 get this session data from the server and sets the parameters of the ProxyInfo extension for the link between proxy_2 and server. It then completes the handshake with proxy_1. This proxy_1 then adds the ProxyInfo data into the session related to the link between proxy_1 and proxy_2. The session data now contains information about two links viz. between proxy_1 and proxy_2 and between proxy_2 and server. This information is then sent over to the client when proxy_1 completes its handshake. In the end the client has now got all the information required about the two links and the proxies which are in between the connection from itself to the server. The client can make the decision of whether to continue with the connection or to break it depending on the information it has. The extension is designed in such a way that it is scalable to any number of proxies defined by the macro MAX_PROXIES. Taking into consideration the network overhead, at any given instance only extension data relevant upto the current proxy is available in the extension without any placeholders.

Chapter 6

Implementation Details

The extension is implemented in C by adding an extension named ProxyInfo to the gnutls open source library. The comprising data structure and APIs that we have implemented and exposed are described in detail as follows,

6.1 *ProxyInfo* structure

The *ProxyInfo* structure is defined in the lib/ext/*proxyinfo.h* file. The contents of the *ProxyInfo* structure are:

```
gnutls_cipher_algorithm_t   cipher_algo;
gnutls_kx_algorithm_t       kx_algo;
gnutls_mac_algorithm_t      mac_algo;
int ip_addr;
int mac_addr;
```

gnutls_cipher_algorithm_t: It is an enumeration of different symmetric encryption algorithms. 3DES in CBC and CFB mode, AES in CBC and CFB mode with 128-bit keys and 192-bit keys, blowfish in CFB mode are some of them.

gnutls_kx_algorithm_t: It is an enumeration of different key exchange algorithms. RSA, DHE-RSA, Anon-DH, etc are some of them.

gnutls_mac_algorithm_t: It is an enumeration of different Message Authentication Code algorithms. HMAC-MD5, HMAC-SHA 224, 256, 384 and 512, etc are some of them.

ip_addr: It stores the IP address of the proxy as an integer. This can be used as a criteria to check if or not the particular IP address is blacklisted. The client can make use of a blacklist of IP addresses containing malicious or untrusted proxies and validate the ip_addr information before deciding on continuing with the handshake.

mac_addr: It stores the mac address of the proxy as an integer. As with the *ip_addr*, this can be used to check if or not the particular mac address is blacklisted. This increases the client's knowledge of the intermediate proxy.

In addition to this, the extension also makes use of *proxy_cnt* which gives the client application the number of proxies which lie in between the client and the server.

6.2 ProxyInfo extension APIs

The APIs necessary for setting and retrieving the proxy specific information by populating the ProxyInfo structure are defined in the file lib/ext/ProxyInfo.c A detailed explanation of the functions is given below:

*6.2.1 int gnutls_proxyinfo_set (gnutls_session_t session, gnutls_server_name_type_t type, const void *name, size_t name_length, void *data, int data_length, int proxy_id)*

This API allows the proxy to bind the information about itself to the current TLS session, it is done by calling the get_proxy_info function which sets the values to the ProxyInfo and then binding the structure to the session passed in the parameter. This function also accepts an argument data which is basically data passed on from the proxy implementation about information it had received the other proxies in the chain until the server.

6.2.2 gnutls_ProxyInfo_ext get_proxy_info(gnutls_session_t session)

This API gets the information pertaining to the session passed as argument and sets it in the ProxyInfo structure.

*6.2.3 int gnutls_proxyinfo_get (gnutls_session_t session, void *data, size_t * data_length, unsigned int *type, unsigned int index)*

This API allows the intermediate proxies as well as the client to get the proxy information which is packed as successive bytes of data and passed as reference through the data buffer. The variable data length is the length of the data which can be used to retrieve the necessary part of data.

6.2.4 *int _proxyinfo_recv_params (gnutls_session_t session, const opaque * data, size_t data_size)*

This function is triggered when an extension data is received by an entity and our implementation breaks the received extension data and sets in them in the local ProxyInfo structure.

6.2.5 int _proxyinfo_send_params (gnutls_session_t session, gnutls_buffer_st data)*

This function is triggered when a new extension data is bound to the session. Our implementation packs the ProxyInfo extension structure into a data buffer and sets it to gnutls_buffer which can be retrieved by the other participating entity in the session.

*6.2.6 int _proxyinfo_pack (extension_priv_data_t epriv, gnutls_buffer_st * ps)*

This packs the data into the Buffer stack in case of a session is paused.

*6.2.7 int _proxyinfo_unpack (gnutls_buffer_st * ps, extension_priv_data_t * epriv)*

This unpacks the data on session resumption and sets it back to the session instance.

6.3 Other APIs

*6.3.1 int get_ip_int(char *p)*

This function retrieves the IP address of the current entity and transforms it into a space efficient bit separated single integer and returns it to be used by the other extension API.

6.3.2 char get_ip_str(int ip)*

This function returns the IP address in a human readable form when given an integer parameter. This was a helper function basically for our own interpretation of the values being passed and was not part of the extension implementation.

Chapter 7

Conclusion

We have defined a new extension which is not supported by the current GNUTLS library by defining its data structure and exposing APIs to make use of it. As a proof-of-concept we have also implemented a basic client-proxy-server setup that makes use of the extension by setting the relevant data pertaining to its corresponding session which is finally available for the client which can decide on completing the handshake based on its priorities.

A possible future work on this would be to include authenticity in the ProxyInfo structure by including the signature of the proxies as part of the structure. Another possible improvement to our implementation is to handle the error mechanism in case of session pauses and resumptions

Chapter 8

References

[1] http://www.gnu.org/software/gnutls/

[2] http://en.wikipedia.org/wiki/Transport_Layer_Security