University of Amsterdam

System & Network Engineering

Reliable client-server connections

Research Project 2

July 5, 2013

Authors:
Thijs Rozekrans thijs.rozekrans@os3.nl
René Klomp rene.klomp@os3.nl

Abstract

The current usage of TLS relies on centralized certificate authorities which poses a single point of failure and introduces costs for signing of certificates. Within this research several existing techniques are used to build a TLS tunneling daemon that validates TLS certificates in a decentralized way. DANE will be used to validate domain certificates by matching them to the certificates stored in DNS. User certificates will be validated using a LDAP server as a PGP key server. By matching the certificates stored in this LDAP server, the client identity can be validated as well. Combining both techniques in a single daemon will allow existing applications, by using small library, to make use of the daemon and establish a reliable and secure TLS connection.

Contents

1. Introduction

Several techniques are available to establish a secure connection between clients and servers. The security is guaranteed by the integrity, confidentiality and authentication mechanisms within TLS. Problems with the current usage of TLS is that it relies on centralized certificate authorities which poses a single point of failure[19] and introduces costs for the signing of certificates.

All techniques are available to create a decentralized authentication solution for both clients and servers in which a reliable and secure client-server connection can be established. Currently no practical implementations exist.

With the introduction of the DNS security extensions (DNSSEC)[1] there is authentication and integrity protection of DNS data. The same technique can be used to store and sign keys and certificates that are used by TLS, this technique is called DANE[11]. Another technique to consider is publishing CERT records [12] for client-side certificates in DNS or another possibility is to store the certificates in LDAP[28] and request a specific certificate. These mechanisms could be used by any service that wants to validate a user's identity or pseudonymity without the need of a CA to sign the certificate. Since TLS supports this, it would even be possible to use PGP certificates here[15].

Combining the techniques mentioned above, client and servers can authenticate each other and establish a secure connection where the data is encrypted and integrity can be guaranteed. This can all be done in a decentralized way using PGP keys and certificates.

The goal of this research is to create a mechanism based on TLS that fulfills the checking of client and server certificates in a decentralised way and establishing a secure connection.

1.1. Research Question

This research will focus on creating a secure connection between a client and a server, which results in the following research question:

> *"How can current techniques be used to validate the identity of both client and server, using a TLS connection, in a decentralised way?"*

In order to completely answer the research question, it can be divided into the following sub-questions:

- What security techniques are available that support the research goal?
- How can client certificates be validated, while guaranteeing privacy?
- How can the security level be validated?

1.2. Related Work

A lot of research and implementation exists on server security (authentication and integrity). However less research is done on the authentication of client using certificates [8] and especially PGP certificates. All the techniques needed for client authentication using PGP certificates are available but not yet implemented in combination with TLS and LDAP or DNS.

2. Background

In the following sections a description is given of the techniques that where used and combined within this research. The purpose of the description is to provide the reader the information to fully understand how the implementation works.

2.1. Transport Layer Security

The Transport Layer Security (TLS) is the successor of Secure Socket Layer (SSL) and is currently at version 1.2. The goal of the TLS protocol is to provide authentication, confidentiality and integrity independently of higher protocols [6]. Any higher protocol can be implemented on top of TLS to add security.

To verify the identity of the communicating parties, authentication is needed. This is done by validating the certificate of both the client and server. This way the client can verify that the server is who it claim to be and vice versa.

In order to guarantee the integrity of the messages, TLS has a message framing mechanism and signs each message with a Message Authentication Code (MAC). The MAC algorithm is a hash function and the keys used are negotiated by both connection peers. This can be seen as a checksum of the messages in which the integrity can be validated.

To encrypt the data being exchanged between the client and the server both have to agree upon a ciphersuite[1]. TLS is based on public key cryptography (also called asymmetric key), this allows the client and server to establish a shared key without prior knowledge of each other.

The TLS protocol can be divided into two protocols, the handshake and the record protocol.

TLS Handshake protocol is used for the negotiation of the encryption algorithm and cryptographic keys before any data is sent by the application. It starts with the authentication in which at least one of the parties need to authenticate based on public key cryptography. After authentication the client and server negotiate a shared key, the key can not be eavesdropped or modified by for example a Man-In-The-Middle attack.

The TLS Record protocol makes sure the connection between the client and the server is private and reliable. It provides symmetric cryptography which is used for data encryption. The keys used for the cryptography are generated and negotiated by the handshake protocol. Another property of the record protocol is the integrity check, based on the MAC.

[1] `http://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-3`

In figure 1 a flowchart of a TLS connection is shown. Since TLS runs on top of TCP, it first need to complete the three-way handshake. After the TCP connection is established, the client sends a *ClientHello* message which contains a number of specifications (TLS version, supported ciphersuites and other TLS options). The server responds with a *ServerHello* containing the chosen version, cipher etc. The server also sends its certificate and in case the client has a certificate, the server sends a certificate request. With the *ServerHelloDone* message the server indicates that it finished the handshake.

The client replies to the request by sending its certificate and if the client finished verifying the certificate, the client generates a new symmetric key. The key is encrypted with the servers public key and sent to the server, with the *ChangeCipherSpec* the client tells the server to switch to encrypted communication from now on.

The server decrypts the symmetric key and checks the integrity by validating the message MAC and then sends the encrypted *Finished* message. From this point on, all the data sent by the application is being encrypted before it is transferred between the client and server.

Assumed is that TLS is running on top of TCP, however TLS protocols exist for User Datagram Protocol (UDP) and Stream Control Transmission Protocol (SCTP). The DTLS protocol allows private communication to take place for datagram protocols [21]. Since UDP is unreliable and packets may arrive out of order, this causes a problem for the regular TLS protocol. The DTLS protocol contains mechanisms to handle these problems. This allows a TLS connection to be established based on UDP. SCTP has some additional feature compared to TCP, multiple streams and message oriented which can not be handled by TLS. Therefore an extension is described in RFC 3436 how TLS can be implemented on top of SCTP [13]. Another extension is DTLS for SCTP which is described in RFC 6083 [27].

2.1.1. STARTTLS

STARTTLS is an extension for insecure (unecrypted) connections between client and servers. Using STARTTLS a connection can be upgraded from an unencrypted to an encrypted connection based on TLS. Several application have a STARTTLS extension, e.g. IMAP and POP3[17], SMTP[10], FTP[7], XMPP[22] and LDAP[9].

For example, the communication between a SMTP client and server is normally unencrypted. The STARTTLS extension for SMTP allows the client and server to negotiate on the use of TLS. The client sends the STARTTLS command if it wants to start using TLS. The server responds if it is ready to set up a TLS connection or returns an error. If the server is ready, the client starts sending the *ClientHello* and a TLS connection is established.

2.2. Domain Name System

The Domain Name System (DNS) is a decentralized hierarchical naming system for domains [16]. DNS is invented to provide a user-friendly naming system to access computer systems and Internet services. It is a hierarchical system to translate names of online

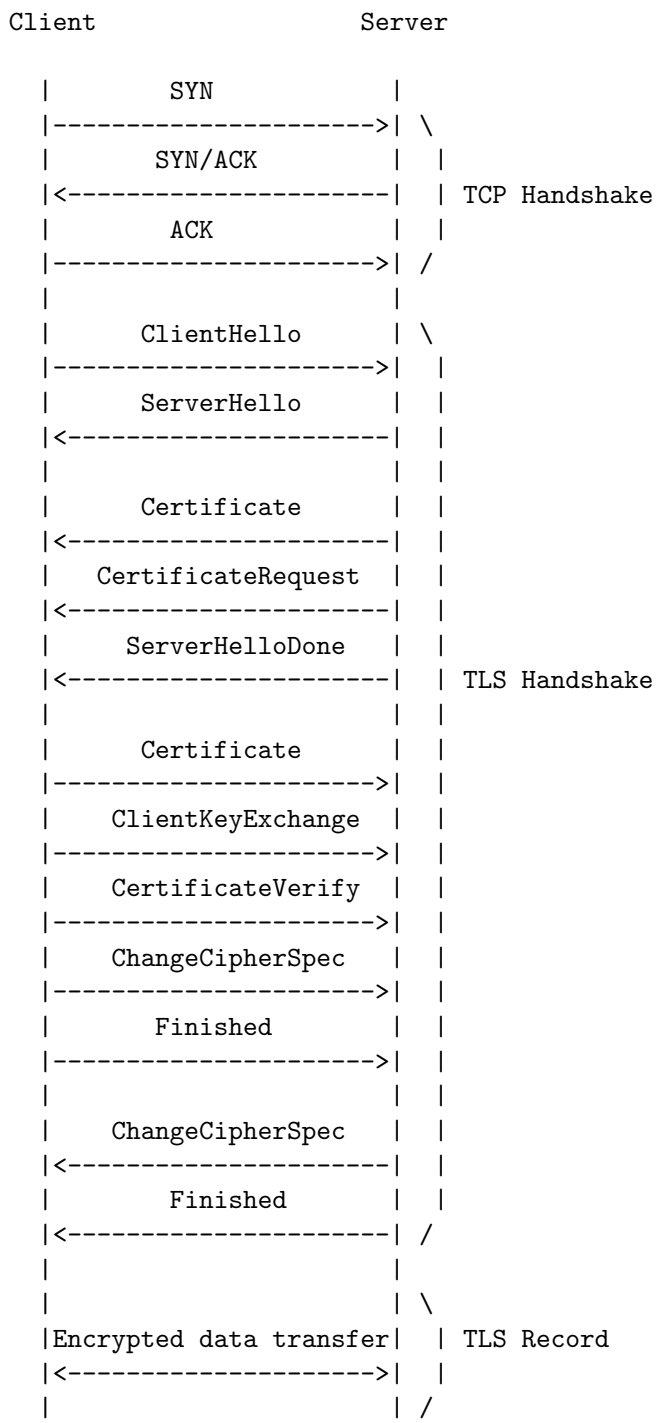

Figure 1: TLS flowchart

resources, e.g. host/domain names, to the corresponding IP address.

There is a set of threats against the domain name system[2], but the security extension of DNS protects against these threats. DNSSEC[1] provides integrity and authentication of existing and denial of existing of the DNS data. Note that DNSSEC does not provide confidentiality of the DNS data, the philosophy behind it, is that DNS data is public information and the returned data by the system is the same for everyone.

DNSSEC data is signed using public key cryptography. The signing happens on a zone level which means that the information in an entire zone is signed. Compared to other protocols the signing in DNS happens up front, since this process is computation expensive. Therefore after a zone is signed, the signature is stored and accessible to DNS servers.

In order to add security to the DNS protocol a chain of trust needs to be created. Authentication is done by digitally signing the DNS data, using Resource Records (RR). Usually one private key sings all names within one DNS zone. Rersource Records are used to build the chain of trust. The chain of trust start at the DNS root in which the Resource Record Signature (RRSIG) contains signatures of the RRsets. The DNSKEY (public key) is used to verify the signatures. The Delegation Signer (DS) contains a hash of a child zone to verify the DNSKEY of that child zone. In figure 2 the chain of trust can be seen for os3.nl. Nowadays the chain of trust starts at the root[2], the keys of the root are known to the resolver. The root delegates the responsibilities to the Top Level Domains (TLD), in this case, the nl zone. The nl zone delegates responsibilities to the domain os3.nl. The domain os3.nl can delegate the responsibilities for possible sub-domains. The advantage of using delegation from a higher domain is that it does not require each domain to trust each sub-domain's public key [18].

2.3. DNS-based Authentication of Named Entities

DNS-based Authentication of Named Entities (DANE) is an extension to the Domain Name System. It provides the ability for domain administrator to advertise the public keys used for TLS connections[11].

A TLS connection makes use of certificates to bind public keys with names. This binding is signed by another key, usually a key of a certificate authority (CA). In this scenario the CA's can issue a certificate for any domain, which can make a TLS connection vulnerable. A CA's who is influenced by for example foreign governments, can handle a "new" certificate and returns a forgery key. This allows those parties to eavesdrop a TLS protected connection[24].

DANE provides the ability to bind a public key with a domain name. It uses the DNSSEC infrastructure to store and sign keys and certificates used with TLS connections. In this scenario there is no need to trust a third party (CA) to sign the binding. This is done by the administrator of the domain.

[2] `http://www.root-dnssec.org/`

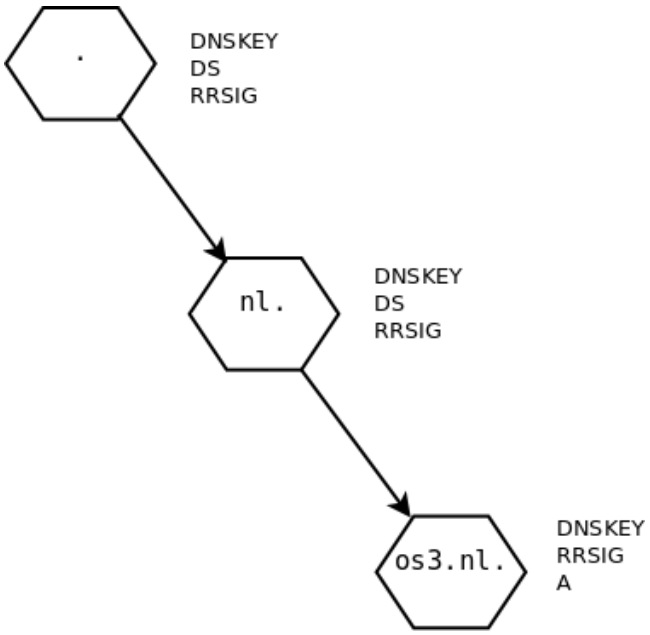

Figure 2: DNSSEC chain-of-trust

2.4. Public Key Infrastructure

PKI can be defined as a set of components to distribute public keys. The public key infrastructure is designed to securely find a public key of the person or service you want to communicate with. It consists of several Certificate Authorities (CA's). CA's are known as trust anchors who generate certificates which consist of the identification and the public key.

The following steps are needed to obtain a digital certificate from a certificate authority. If an user or service wants to obtain a certificate it first needs to generate a key pair. This key pair consist of a private and public key. Now the certificate of the CA is requested. The CA sends it's certificate which consists of the public key and the digital signature (which is singed by the private key of the CA). A certificate request to obtain a certificate is sent to the CA consisting of the public key and personal information (e-mail address, fingerprint, etc.). This request is encrypted by the CA's public key. Next the CA verifies the identity and generates a certificate (binding of public key and identity). The signature of the CA verifies the authenticity of the certificate. Now the CA issues the certificate to the requester.

2.4.1. X.509

The X.509 is a standard defined by the ITU for the public key infrastructure (PKI). It specifies the format of certificates. The certificates issued by the PKI (CA's) all have the X.509 format. A variant of the X.509 certificate which is not signed by the PKI is called

a self signed certificate. In this case the issuer and the identity of the certificate holder are the same. Now the certificate needs to be verified by a third party before it can be trusted. In PGP for example the certificates are signed by the other communicating party, in this case everyone can sign each others certificate.

A X.509 certificate consist of the following information [25]:

- Version - Indication of the version used;
- SerialNumber - Unique identifies the certificate;
- Signature - Identifies the algorithm used to compute the digital signature;
- Issuer - Name of the CA;
- Validity - Expiration date;
- Subject - Name of the owner of the keys certified;
- SubjectPublicKeyInfo - Consist of the identification of the algorithm used and the public key;
- IssuerUniqueIdentifier - Identification of the issuer;
- SubjectUniqueIdentifier - Identification of the owner;
- AlgorithmIdentifier - This field repeats the signature field;
- Encrypted - Field containing the signature.

Extensions exist on the X.509 certificate format but these field are not often used in practice [25].

2.5. Pretty Good Privacy

The Pretty Good Privacy (PGP) protocol is often used to encrypt e-mails. PGP is not just for mail, but it also performs encryption and integrity of files[5]. It uses public keys for personal keys. Compared to other protocols which are using public keys, PGP differs how public keys are certified and how certificate chains are verified. PGP enables users to decide which keys to trust instead of letting a certificate authority or infrastructure decides whom you should trust. This could be a challenge finding a public key, but people publish their PGP fingerprints (cryptographic hashes of public keys) on for example business cards, e-mails etc. This way a web op trust can be created.

If a PGP user wants to sent encrypted data, PGP generates a session key which is a random number used only once. The data is encrypted with the session key, the session key is encrypted with the public key of the recipient. The encrypted session key is sent along with the encrypted data. The recipient uses his private key to decrypt the session key, which is then used to decrypt the data.

Another property of public key cryptography is the ability to use digital signatures. Digital signatures allows the recipient to verify authenticity, integrity and non-repudiation. In order for a user to create a digital signature for the data, the data is encrypted with the private key instead of the public key used in encryption. The recipient can decrypt the data with the public key of the sender. Within the public key environment there is no authentication mechanism in place. Therefore users using a public key to encrypt data have to make sure the public key belongs to the intended person and is not a forgery.

As the X.509, the PGP certificate has a specific format including the following information [25]:

- Version number - Indicates the version of PGP used to create the key;
- The public key of the owner - The public key of the key pair;
- The certificate owner information - The information consist mainly of the name of the owner;
- Certificate owner digital signature - The digital signature, signed by the owner;
- The validity period - Indicates the certificate expiration date and time;
- Preferred encryption algorithm for the key - Indicates the encryption algorithm preferred by the certificate owner to encrypt the data.

2.6. Lightweight Directory Access Protocol

The predecessor of LDAP[28, 23], the Directory Access Protocol (DAP), was too large and too complex to run on smaller machines, therefore LDAP was designed. The goal of LDAP was to provide low-overhead access to the X.500 directory.

All data on the LDAP server is stored in object and attributes, this data can be accessed with the following operations: search, add, delete, modify, modify RDN, bind, unbind, and abandon. LDAP is often combined with an e-mail directory which allows employees to access contact information of other employees within the same company.

Not only contact information can be stored in LDAP, an extension exists to store certificates[3]. This OpenPGP extension is not present in the attribute library of LDAP by default and therefore needs to be installed. The extension adjusts the LDAP schema allowing LDAP to store certificates. The LDAP server is now called a PGP keyserver, allowing PGP keys to be stored and retrieved by clients. Using the SRV record in DNS, the client is able to locate the LDAP/PGP keyserver. On the keyserver a look-up can be performed to find the PGP key of a specific person. This could also be extended with authentication mechanisms to prevent unauthorized access to this private information.

[3] `http://lists.gnupg.org/pipermail/gnupg-users/2006-February/028058.html`

3. Design Considerations

The implementation starts with a TLS connection based on a secure communications library called GnuTLS, for which a Python wrapper is available[4]. The reason to choose for the PyGnuTLS fork in stead of the original python-gnutls library available from the Python website[5] is the support for Pretty Good Privacy (PGP) certificates. Below several design considerations are discussed.

3.1. PGP or X.509

There are some differences between PGP and X.509 certificates[20]. Compared to a X.509 certificate which supports only one signature, a self-signed PGP certificate can contain multiple signatures. The validation of a X.509 certificates is always done by a certificate authority. Using PGP it is up to the user to validate another PGP certificate. The management of keys also differs, PGP users manage their own keys while X.509 certificates are managed by a CA. Another advantages of PGP is not being dependent of a CA to revoke certificates. If the user feels there is tampered with his certificate it can be revoked by himself. The overall design of PGP suits the requirements better and fits in the decentralized design of the solution.

3.2. Storing Private Keys

The certificates are usually stored as a file on the server. This could be a security risk in case a server gets compromised or infected by malicious software. In that case the attacker could have full access to the server and can steal the private key and the digital certificate. There is an alternative to storing certificates, this can be done using a Hardware Security Module (HSM). This device is special designed to manage cryptographic keys and providing encryption, decryption, authentication, and digital signing of data. Generally this is an expensive device and therefore a software alternative is created, called SoftHSM[6]. The implementation of SoftHSM is a cryptographic store accessible through the general PKCS#11 interface.

3.2.1. Server Certificates

Within this research both client and server make use of PGP certificates to authenticate to each other. As described in section 2.4, currently CA's are used to sign the certificates. In the past there have been issues with the security of certificate authorities[19], which make all the certificates signed by the organization invalid. Another potential issue with CAs could be that the organization is influenced by the government, allowing them to decrypt all the secure communication[24].

When making use of PGP all the certificates are signed by the users themselves creating the web of trust. In this case the operators and users are responsible for the signing

[4] `https://gitorious.org/pygnutls`
[5] `https://pypi.python.org/pypi/python-gnutls`
[6] `http://www.opendnssec.org/softhsm/`

of the certificates. There is therefore no need for a central CA to sign the certificate, making this solution decentralised.

Since CA's are part of a PKI which allows public key look-ups to verify communicating to the correct party. This infrastructure is not available when making use of PGP, making it more difficult to look up public keys. A solution available described in section 2.3 called DANE, allows public key information to be stored in DNS.

3.2.2. Client Certificates

The user certificates can also be stored in DNS using CERT resource record (RR) . The CERT resource records are defined so that such certificates and revocation lists can be stored in the DNS [12].

DANE as described in the latter section can not be used in combination with user certificates, because it requires a domain name in order to identify the server. Users are not identified based on a domain name, the identification is based on an e-mail address.

However there are some privacy concerns to storing certificates in DNS. Since everyone can make use of this Internet service, everyone can gather personal information about the users. This is solution is very vulnerable for spam and scam. When using DNSSEC this is even more vulnerable because of the so called zone walking, which introduces the ability for a hostile party to enumerate all the names in a zone. NSEC3[14] tries to solve this by hashing the domain names, but also this technique is not fully secure[4] and it is not recommended to hide data in a zone. All data should be assumed public.

Besides that the DNS system in not meant for contact information and does not scale in an environment with a large amount of users. For administrative reasons it is better not to store this in DNS since this is not modifiable by users and in larger companies probably managed by another department.

Another solution available to store certificate is to store them in LDAP. The advantage of LDAP compared to DANE, is the availability of an authentication mechanism. This prevents unauthorized access to the detailed contact information. To retrieve the certificate from LDAP system, a baseDN and a search filter can be selected. Based on the e-mail address or the UID of the certificate, the certificate can be retrieved.

3.3. Daemon or Library

One of the design considerations was to implement this solution in a daemon. There are several reasons why to choose a daemon instead of implementing the solution in a library. The reason against a daemon is that the GnuTLS library was already available and this library could easily be extended to fulfill the requirements. However implementing this solution within a daemon has more advantages.

The general advantages of a daemon are the possible extensions which can be implementing easier within a daemon. A forwarding mechanism towards back-end server can only be implemented within a daemon. Another feature that can be implemented in a daemon is a caching mechanism which could potentially cache the certificates or results from LDAP and DNS queries. Access control and access to private keys can better be

managed within a daemon. Since only one process, the daemon, which is not directly contactable from the Internet, needs access to the certificates this provides an extra layer of security for protecting the private keys. The daemon could be used by multiple programming languages by creating a very small library that contacts the daemon in that particular language. This is not possible if this solution is implemented within a language specific library. In case of a language specific library, a full wrapper for all functions has to be made to support it within another language.

3.4. Programming Language

Since the decision is made to create a daemon, the next choice to make is the programming language to use. If there was chosen to extend the current library, the only choice would have been the C-language, but since the is not necessary anymore the programming language can be chosen more flexible. The final decision was made for python. A simple and readable language that allows quick development and easy modifications in the future. It contains, or is extensible with, all functionality needed for this project and is fast enough for our purposes. The most computational expensive calculation of encryption and decryption takes place in the GnuTLS library which is written in C.

4. Implementation

In figure 3 the establishment of the TLS connection and the relation between the different components can be seen. In section 4.1 all the steps are described during the setup of the connection. The details about the LDAP and DANE implementations will shortly be discussed in sections 4.2 and 4.3. Finally a description about the testing applications in section 4.4 and an assumption about the performance can be found in section 4.5.

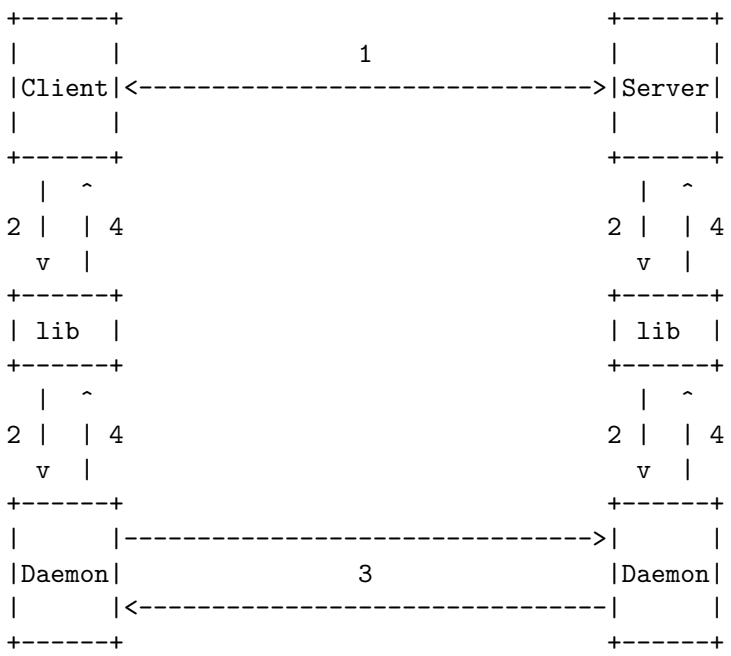

Figure 3: Setup of the reliable client-server connection.

4.1. Invoking the daemon

The daemon can be invoked from any application via an intermediate library, made for that language, or directly. This intermediate library is also known as a stub-library. The client starts by setting up a TCP connection with its peer (1). This TCP connection can then be used non-encrypted and later be upgraded to TLS using STARTTLS, or directly using TLS. In both cases the client application invokes the daemon by sending the file descriptor of the socket together with a command and some flags as described in 4.1.1 through an UNIX socket to the daemon (2). The command can be `start-tls` or `recv-tls`. Based on this command and file descriptor the daemon recreates the socket object and starts the TLS handshake relatively as the client or as the server. After a successful handshake, the certificate received from the peer is validated based on the uid of the certificate. The daemon chooses the validation method. Whenever the uid contains an e-mail address it is assumed that the certificate is an user certificated and

thus it should be validated via LDAP (section 4.2) otherwise it is assumed to be a domain certificate and the certificate is checked using DANE (section 4.3). When the validation succeeds the TLS connection setup is done (3) and the daemon continues by creating a new socket pair. The file descriptor of one of the sockets is subsequently returned to the application that invoked the daemon (4). The application then uses this socket object to communicate with the peer via the daemon. The daemon subsequently proxies all traffic and encrypts/decrypts it. When the application wants to close the connection it sends a `quit` command to the daemon and the daemon then tears down the TLS connection by closing all sockets. The traffic flow as shown in figure 4, without the validation of the certificates, can also be seen with Wireshark[7].

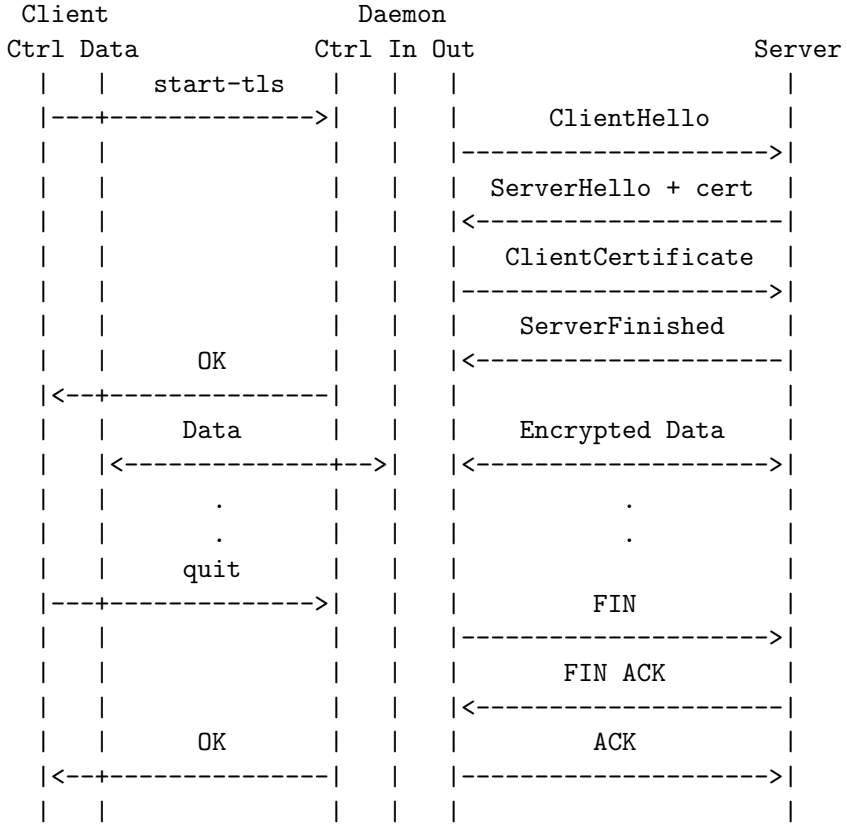

Figure 4: Daemon connection flowchart

4.1.1. Flags

As mentioned in the previous section the application can pass some flags to the daemon, these flags control which functions are *not* used. There was explicitly chosen to make

[7] `http://www.wireshark.org/`

the flags in a negative form so that the user has to disable security functions explicitly. This way nothing can be forgotten by accident.

The flags that are currently implemented are `no-dnssec`, `ignore-bogus`, `no-dane` and `no-ldap`. The `no-dnssec` flag makes sure that the DANE and LDAP implementations also work when no DNSSEC is available. It still checks for DNSSEC and if an invalid DNSSEC record is found this can be ignored using the `ignore-bogus`, otherwise it aborts on a bogus DNSSEC record. The latter two, `no-dane` and `no-ldap` disable the checking of certificates in respectively DANE and LDAP.

4.1.2. Responses

After a successful handshake and validation in DANE or LDAP the daemon returns a file descriptor and status message to the user. This status message can be `OK` followed by the identity of the validated user or `ERR` followed by an error code and a descriptive message. Currently errors that can be returned are that DNSSEC is invalid, no user was found in LDAP or that no valid TLSA record can be found.

4.2. LDAP

A user certificate is looked up in LDAP. To find the address of the LDAP server that needs to be queried the daemon starts by doing a look-up for a SRV record of the domain `_pgpkey-ldap._tcp.<domain>`. The returned records are subsequently based on their priority resolved to the address of the LDAP server and the LDAP server is queried using this IP address and the port numbers as found in the SRV record. The certificate of the LDAP server should be validated using DANE.

Whenever an entry is found matching the search query, the fingerprints of the certificates in the user record are matched against the fingerprint of the peers certificate. If one of them matches it means that the identity is valid and thus that the peer supplied a valid certificate. When no entry is found the daemon returns an error.

4.3. DANE

Since the PyGnuTLS library does not support DANE this has been implemented manually. This also gives the ability to have a more fine-grained control over what is happening and to act on exceptions. DANE is just a simple DNS request with the TLSA type trying to resolve `_<port>._<protocol>.<domain>` where protocol is e.g. TCP or UDP. The result of this query is checked for valid DNSSEC and if that is valid or disabled using the `no-dnssec` flag, the hash that was found is compared to the correct hash, based on the *matching type* field, of the peers certificate. If the match DANE succeeds, but if they do not match the daemon return an error response.

4.4. Applications

To test the daemon a simple client and server application is used. The client sends the `start-tls` command to the daemon as soon as the TCP connection is set up and

the server sends the `recv-tls` to the daemon as soon as a client has connected. After setting up the TLS connection both can communicate with each other over TLS via the daemon.

To further show the capabilities of the daemon, a Stunnel[8] like application has been created to be able to tunnel unencrypted application traffic, e.g. Telnet, via the daemon over TLS. This situation is shown in figure 5. The Telnet client communicates via two TCP forwarders to the Telnet daemon. As soon as the connection has been set up, the TCP forwarders hand it over to the daemon to upgrade the connection to TLS after which the Telnet traffic is sent encrypted over the network.

With a single line of code every application would be able to upgrade its connection to a secure and reliable TLS connection.

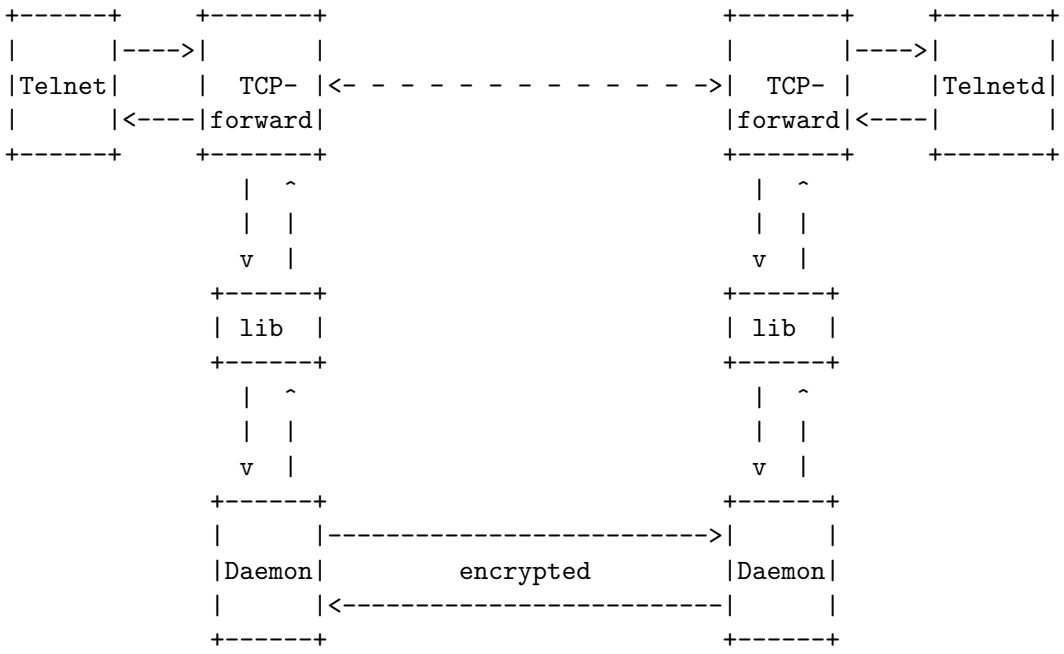

Figure 5: Forwarding mechanism

4.5. Performance

No measurements have been done on the performance of the daemon, but one could assume that these performance penalties would not be very high. Since the encryption and decryption is the most time consuming process in a TLS encryption and this is all done within the GnuTLS library, the performance penalty of all traffic going trough the daemon is negligible. Only during the handshake phase the performance is effected in a negative way by the DANE and LDAP functionality. These functions gives that one or

[8] `http://www.stunnel.org`

more DNS look-ups have to be done and in the case of a user certificate in LDAP also a LDAP look-up has to be done. The latter can delay the handshake process significantly when timeouts occur and fallback servers have to be contacted, but in any other case the delay in the handshake process will probably not be worrisome.

To further improve the performance of the implementation caching could be used. In order to accelerate re-connections, LDAP and DNS queries as well as succeeded TLS connections could be cached. This will also improve the performance within environments having large amount of connections.

Besides this, certificates could be pinned. This technique allows a host to associate another service or host with a certificate or public key, this could improve the performance. The changes in the certificate(s) can be detected, preventing potential security risks [26].

5. Logic

This research focused on a secure connection between a client and a server. To show the level of security of the implementation, mathematical logic is used. This type of logic is also called semantics. Below the equations and the explanations can be found.

The starting point of the security within this implementation is the untrusted ID who connects to one of the services. The untrusted ID can prove if the identity is correct by showing the ability to use the private key.

5.1. DNSSEC

To verify that the domain name belongs to the service the client wants to connect to DNSSEC is used. DNSSEC as described in section 2.2, uses a chain of trust to validate the validity of the domain name. A higher level domain signs the key of child domains this way the child domain is part of the chain of trust. Based on this chain of trust the validity there can be assumed that the id belongs to the correct service.

$$((P(key, DNS + id) \wedge S^*(DNSroot, DNS + id) \tag{1}$$

x = dnskey
y = domain name
$P(x, y) = x$ is present in y
$S^*(x, y)$ = is the transitive closure of $S(a, b)$ which means "a signs for b".

If the ID is present in the key and in DNS, and this applies up to the root of the DNS system then the ID (of the domain name) can be trusted. More reasoning about DNSSEC can be found in [3].

5.2. DANE

The identity of the server can be verified based on a certificate. DANE enables those certificates to be stored in DNS. This way the certificates can easily be accessed by the client connecting to the server. The identity in DANE is proven in equation 2.

$$\exists r(S(r) \wedge H(r, k)) \Leftrightarrow T(k) \tag{2}$$

k = key
r = TLSA record
$S(x)$ = DNSSEC signs x
$H(x, y) = x$ matches hash of y
$T(x) = x$ is trustable

For all keys, if there exist a TLSA records for which the DNSSEC signature is correct and the hash matches the hash of the key, then the key is trustable and thus the ID of the certificate can be trusted.

5.3. LDAP

In the case the untrusted ID is a user ID the daemon looks up the ID on a LDAP key server. Whenever the public key of the user matches the public key that is stored on the queried LDAP server there is proven that the key supplied by the user is valid and that the that the ID relating to the key is the users real ID and can be trusted.

To prove that the key as stored in LDAP is correct there is relied on the fact that the connection between the daemon and the LDAP server uses TLS and thus that the identity of the LDAP server and the integrity of the received data can be validated. DNSSEC is used as described in section 5.1 to validate the DNS lookup of the IP address based on the SRV record that was also retrieved from DNS and validated with DNSSEC. Based on those implications there can be stated that if validation is correct that the DNSSEC is correct. Their can also be safely relied on the returned value from the LDAP server and thus validate the clients ID.

$$\exists l(H(l,k) \wedge I(l) \wedge \exists a(R(l,a) \wedge \exists d(S(d) \wedge Q(d,a) \wedge \exists s(S(s) \wedge Q(s,d))))) \Leftrightarrow T(k) \quad (3)$$

k = key
l = LDAP server
a = IP address
s = SRV record
d = Domain name
$H(x,y) = x$ has key y
$I(x) = x$ is reachable over TLS
$S(x)$ = DNSSEC signs x
$R(x,y) = x$ has address y
$Q(x,y) = x$ resolves to y
$T(x) = x$ is trustable

For all keys applies that, if there exists a LDAP server which:
holds the key, is reachable over TLS and there exists an IP address which:
points to the LDAP server and there exists a domain name: that is signed by DNSSEC and resolves to the IP address and has a SRV record which:
is also signed by DNSSEC and which is resolvable to d. Then the key is trustable.

5.4. Combined

If equation 2 is called $D(x)$ and the equation of the previous section about LDAP, equation 3, is taken as $L(x)$, they can be combined to give the full logic implication of the daemon. The above equations combined results in the following equation:

$$\exists k_1, k_2((D(k_1) \vee L(k_1)) \wedge (D(k_2) \vee L(k_2))) \Leftrightarrow C(k_1, k_2) \quad (4)$$

k_1, k_2 = the keys used by the peers
$D(x) = x$ has a valid DANE record

$L(x) = x$ has a valid LDAP entry
$C(x, y) =$ a reliable TLS connection can be made using trusted keys x and y

For the keys (k_1, k_2) used by the peers, k_1 has to be validated using DANE or LDAP *and* k_2 has to be validated using DANE or LDAP. If both keys can be validated using one of the two validation methods than a reliable TLS connection can be made using trusted keys k_1 and k_2.

6. Conclusion

During this project there is shown that with existing techniques it is possible to create a daemon based on TLS that fulfills the checking of client and server certificates in a decentralised way. The domain certificates are validated using DANE and user certificates using LDAP in order to guarantee the users privacy. This solution is implemented within an open-source proof of concept.

During the design and implementation phase possible future extension are taken into account. By using only a small library as a layer between the daemon and the application, existing applications written in different programming language can make use of the daemon by only converting this library. This allows the existing application to establish a connection based on TLS daemon.

The security of this implementation is verified in term of mathematical logic. The logic has showed that the untrusted identity can be validated by showing the ability to use the private key.

There is illustrated and proved that the implementation does work with existing applications. A TCP proxy application has been created to tunnel all application network traffic securely over TLS via the daemon. By taking Telnet as an example, the daemon allows such insecure applications to be more secure by using TLS.

6.1. Future Work

In this project a proof of concept is created for a reliable client-server daemon. To further develop this daemon and make it usable in production environments consists of implementing a caching mechanism. LDAP and DNS queries as well as succeeded TLS connections could be cached this to accelerate re-connections. The daemon could also be extended to pin certificates, to further improving the performance.

Current the implementation only supports a TLS connections based TCP. Extension to the implementation could be created to support other protocols, e.g. DTLS for UDP and SCTP based connections.

Currently the keys and certificates are stored in a directory on the server and client and their location is hard-coded in the daemon. The security and flexibility of storing certificates can be improved by implementing the retrieval of certificates from a (Soft)HSM or a token behind a PKCS#11 application programming interface (API).

Depending on the programming language of the application, the library could be rewritten in that specific language to invoke the daemon. To allow the daemon to work for all the programming languages, multiple libraries could be created.

References

[1] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), Mar. 2005. Updated by RFCs 6014, 6840.

[2] D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833 (Informational), Aug. 2004.

[3] K. Babu, V. Padmanabhan, and W. Bhukya. Reasoning about dnssec. In C. Sombattheera, A. Agarwal, S. Udgata, and K. Lavangnananda, editors, *Multi-disciplinary Trends in Artificial Intelligence*, volume 7080 of *Lecture Notes in Computer Science*, pages 75–86. Springer Berlin Heidelberg, 2011.

[4] J. Bau and J. C. Mitchell. A security evaluation of dnssec with nsec3. In *Network and Distributed Systems Security (NDSS) Symposium. The Internet Society*, 2010.

[5] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), Nov. 2007. Updated by RFC 5581.

[6] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[7] P. Ford-Hutchinson. Securing FTP with TLS. RFC 4217 (Proposed Standard), Oct. 2005.

[8] A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. E. Seamons, and B. Smith. Advanced client/server authentication in tls. In *Network and Distributed System Security Symposium*, pages 203–214, 2002.

[9] J. Hodges, R. Morgan, and M. Wahl. Lightweight Directory Access Protocol (v3): Extension for Transport Layer Security. RFC 2830 (Proposed Standard), May 2000. Obsoleted by RFCs 4511, 4513, 4510, updated by RFC 3377.

[10] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard), Feb. 2002.

[11] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), Aug. 2012.

[12] S. Josefsson. Storing Certificates in the Domain Name System (DNS). RFC 4398 (Proposed Standard), Mar. 2006. Updated by RFC 6944.

[13] A. Jungmaier, E. Rescorla, and M. Tuexen. Transport Layer Security over Stream Control Transmission Protocol. RFC 3436 (Proposed Standard), Dec. 2002.

[14] B. Laurie, G. Sisson, R. Arends, and D. Blacka. DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. RFC 5155 (Proposed Standard), Mar. 2008. Updated by RFCs 6840, 6944.

[15] N. Mavrogiannopoulos and D. Gillmor. Using OpenPGP Keys for Transport Layer Security (TLS) Authentication. RFC 6091 (Informational), Feb. 2011.

[16] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (INTERNET STANDARD), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

[17] C. Newman. Using TLS with IMAP, POP3 and ACAP. RFC 2595 (Proposed Standard), June 1999. Updated by RFC 4616.

[18] R. v. R. Paul Brand, Rick van Rein and D. Yoshikawa. Hardening the internet - the impact and importance of DNSSEC. 2009.

[19] J. Prins. Diginotar certificate authority breach operation black tulip. Fox-IT, November 2011.

[20] N. Prohic. Public key infrastructures–pgp vs. x. 509. In *INFOTECH Seminar Advanced Communication Services (ACS)*, 2005.

[21] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), Apr. 2006. Obsoleted by RFC 6347, updated by RFC 5746.

[22] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), Oct. 2004. Obsoleted by RFC 6120, updated by RFC 6122.

[23] J. Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511 (Proposed Standard), June 2006.

[24] C. Soghoian and S. Stamm. Certified lies: Detecting and defeating government interception attacks against ssl. In *Financial Cryptography and Data Security*, pages 250–259. Springer, 2012.

[25] M. Speciner, R. Perlman, and C. Kaufman. *Network Security: Private Communications in a Public World*. Pearson Education, 2002.

[26] G. Toth and T. Vlieg. Additional certificate verification methods for tls client applications. 2013.

[27] M. Tuexen, R. Seggelmann, and E. Rescorla. Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP). RFC 6083 (Proposed Standard), Jan. 2011.

[28] K. Zeilenga. Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. RFC 4510 (Proposed Standard), June 2006.

A. Source Code

The source code can be found on the public github repository:
`https://github.com/OS3/rp2_68`