BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CONTINUOUS INTEGRATION SYSTEM FOR INTEROPERABILITY OF TLS/SSL LIBRARIES

PRŮBĚŽNÉ TESTOVÁNÍ INTEROPERABILITY KNIHOVEN TLS/SSL

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **FRANTIŠEK ŠUMŠAL**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

ŠUMŠAL, František. *Continuous Integration System for Interoperability of TLS/SSL Libraries*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Smrčka Aleš.

Continuous Integration System for Interoperability of TLS/SSL Libraries

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Aleš Smrčka, Ph.D. The supplementary information was provided by Stanislav Židek and Hubert Kario. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
František Šumšal
March 6, 2017

Acknowledgements

Here it is possible to express thanks to the supervisor and to the people which provided professional help (external submitter, consultant, etc.).

TODO: Stanislav Zidek, Hubert Kario, Jakub Mach, ...

Contents

Chapter 1

Introduction

SSL/TLS is the most widely used technology for securing today's Internet communications. Every application, which trasmits data over the Internet, should implement some kind of encryption and many of these applications use SSL/TLS. But, as the correct way of implementing such encryption can be very tricky, and a simple mistake may have severe consequences, several general-purpose implementations – libraries – were created. These libaries allow the application to use SSL/TLS without having to create their own implementation.

Even though these libraries follow certain standards, which should ensure their inteoperability (ability to communicate with any library which implements SSL/TLS according to the standards), they may contain little deviations or issues which can cause unexpected behavior or even some more serious problems (like denial of service). To avoid, or to at least detect, such issues we can implement tests, which will test the inteoperability between the SSL/TLS libraries. This is one of the main goals of this thesis.

Executing these tests manually for different environments would be ineffective, tedious and not error-prone. Thus, using an automation, which would test various combinations of different library versions on different versions of operating systems is necessary. Such automation – in this case we call it *continuous integration* – is the second main topic of this thesis.

By combining all these features together, we get a powerful system for continuous SSL/TLS library testing. This system can be used for regression detection and ensuring, that interoperability between two given libraries works in a specific environment with a specific combination of settings.

All necessary components for such system are described in the following chapters, starting with a brief description of SSL/TLS protocols and their features in chapter 2.

Chapter 3 describes various continuous integration systems, which were considered for the final solution, and how the implemented solution looks like.

The testing itself, along with used technologies, is discussed in chapter 4. This chapter also includes a list and description of libraries, which were chosen to be tested and why.

During the testing phase several issues were found. Some of them are harmless, others can have a serious impact on the application, which uses given library. Summary of these results is detailed in chapter 5 following a thesis summary in chapter 6.

Chapter 2

SSL/TLS

Secure Socket Layer (SSL) and its successor *Transport Layer Security* (TLS) are cryptographic protocols designed to provide communications security over a computer network. As the latest version of the SSL protocol (version 3.0 [1]) was deprecated in June 2015 [6] it will not be discussed further in this thesis.

Although there are currently three TLS versions – TLS 1.0 [9], TLS 1.1 [10] and TLS 1.2 [11] – this thesis focuses only on the latest two as TLS 1.0 is basically SSL 3.0 with a few differences. Also, there were few cryptographic problems found in TLS 1.0 and later resolved in TLS 1.1 (e.g. BEAST [4]).

2.1 Overview

[[TLS history?]] The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications. The structure of the protocol comprises two layers: the TLS Record Protocol and the TLS Handshake Protocol.

The TLS Record Protocol lies at the lowest level, above some reliable transport protocol (e.g., TCP [1]). This protocol provides security which has following properties:

- The connection is private. Symmetric cryptography is used for data encryption (e.g., AES [2], Camellia, 3DES [3], etc.). The keys for the symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by another protocol (such as the TLS Handshake Protocol). The TLS Record Protocol can also be used without encryption.

- The connection is reliable. Message transport includes a message integrity check using a keyed MAC [4]. Secure hash functions (e.g., SHA-1 [5], SHA-256, etc.) are used for MAC computations. The MAC is used to prevent undetected data loss or data modification during transmission.

The TLS Record Protocol is used for encapsulation of various higher-level protocols. One such protocol is the TLS Handshake Protocol. This protocol allows client authentication and negotiation of necessary properties of the TLS connection, like encryption algo-

[1] Transmission Control Protocol
[2] Advanced Encryption Standard
[3] Triple DES (Data Encryption Standard)
[4] Message Authentication Code
[5] Secure Hash Algorithm

rithm or cryptographic keys, before the data transmission. The TLS Handshake Protocol provides connection security which has following properties:

- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA [6], ECDSA [7], etc.). This authentication is not mandatory, but it is generally required for at least one of the peers.
- The negotiation of a shared secret is secure. Obtaining of the shared secred is infeasible for any eavesdropper or attacker, who can place himself in the middle of the authenticated connection.
- The negotiation is reliable. The negotiation communication cannot be modified without being detected by the parties to the communication. [11]

In addition to the properties above, a carefully configured TLS connection can provide another important privacy-related property: forward secrecy. This property ensures, that any future disclosure or leakage of encryption keys cannot be used to decrypt any TLS communnications recorded or eavesdropped in the past.

TLS supports various combinations of algorithms for key exchange, data encryption and message integrity authentication, which is an important fact for this thesis and can cause severe issues when these combinations are configured or implemented improperly. Along with these combinations, TLS supports many extensions further extending its capabilities and possibilites, which will be described further in this thesis.

2.2 TLS Protocols

As mentioned above, the TLS protocol consists of four protocols – the TLS Record Protocol, the TLS Handshake Protocol, the TLS Changer Cipher Spec Protocol, the TLS Alert Protocol and the TLS Application Data Protocol. In this section we will overview and discuss these protocols in more detail.

2.2.1 TLS Record Protocol

In section 2.1 we briefly described the main function of the TLS Record Protocol, which is encapsulation of protocol data from higher layers. This includes fragmentation, optional compression, MAC application, encryption and transmission of the data. On the receiving side the data is decrypted, verified, decompressed, reassembled and then delivered to the higher layers.

Through the data processing, following TLS data structures are used – `TLSPlaintext`, `TLSCompressed` and `TLSCiphertext`. At the end a TLS record is formed by appending an TLS record header to the `TLSCiphertext` structure.

[[Describe fragmentation, compression and encryption?]]

2.2.2 TLS Handshake Protocol

The TLS Handshake Protocol is the core protocol of TLS which operates on top of the TLS Record Protocol. Its goal is the authentication of communnicating peers and negotiaton of security parameters necessary for establishment or resumption of secure sessions.

[6] Rivest-Shamir-Adleman cryptosystem
[7] Elliptic Curve Digital Signature Algorithm

[[current/pending write/read state]]

The session establishment consists of following steps:

- Protocol version negotiation
- Cipher suite negotiation
- Server authentication and (optional) client authentication using digital certificates
- Exchange of session key information

The actual session establishment using the TLS Handshake Protocol proceeds as follows (see Figure 2.1):

1. The client sends a `ClientHello` message to the server, that includes the TLS version a list of cipher suites supported by the client (in the client's order of preference) and the client's random value, which is used in subsequent computations.
2. The server responds with a `ServerHello` message, including the protocol version, the cipher suite chosen by the server, the session ID, and the server's random value.
3. If the server is to be authenticated, it sends its certificate in a `Certificate` message.
4. A `ServerKeyExchange` message may be sent if the client needs some additional information for the key exchange.
5. If a client authentication is required, the server sends a `CertificateRequest`.
6. Finally, the server sends a `ServerHelloDone` message, to indicate that the hello-message phase of the handshake is complete.
7. If the server has sent the `CertificateRequest` message, the client must send the `Certificate` message containing its certificate.
8. The client sends a `ClientKeyExchange` message. Its content depends on the chosen key exchange algorithm.
9. If the client has sent its certificate to the server, it must also send a digitally-signed `CertificateVerify` message, which explicitly verifies posession of the private key belonging to the client's certificate.
10. The client sends a `ChangeCipherSpec` message to the server, using the TLS Change Cipher Spec Protocol (see section 2.2.3) and copies its pending write state into the current write state.
11. The client sends a `Finished` message to the server under the new write state (with the new algorithms, keys, and secrets).
12. In response, the server sends its own `ChangeCipherSpec` message, copies its pending write state into the current write state and send the `Finished` message under the new cipher spec.

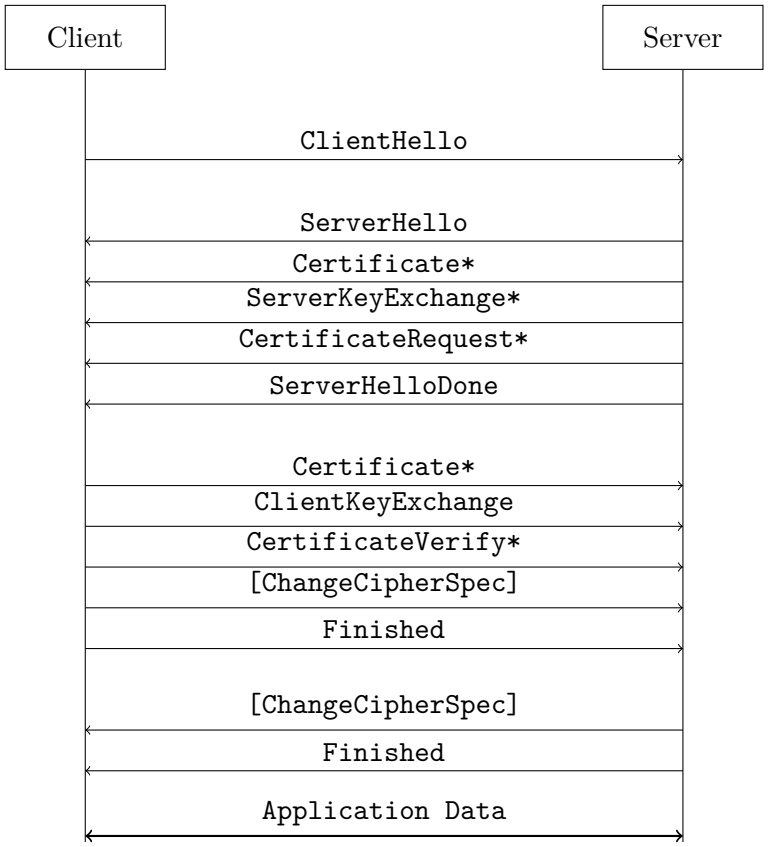

Figure 2.1: Full TLS handshake

Note: * marks messages which are sent only under specific conditions.

At this point the TLS handshake is complete, and the peers may begin to exchange application layer data.

When the client and the server decide to resume a previous session or duplicate an existing one, the handshake can be simplified considerably (see Figure 2.2). The client sends a `ClientHello` message including the ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it sends a `ServerHello` message with the same Session ID value. The client and the server can then directly move to the `ChangeCipherSpec` message followed by the `Finished` message. If a Session ID match is not found in the session cache, the server generates a new session ID and the peers perform a full TLS handshake.

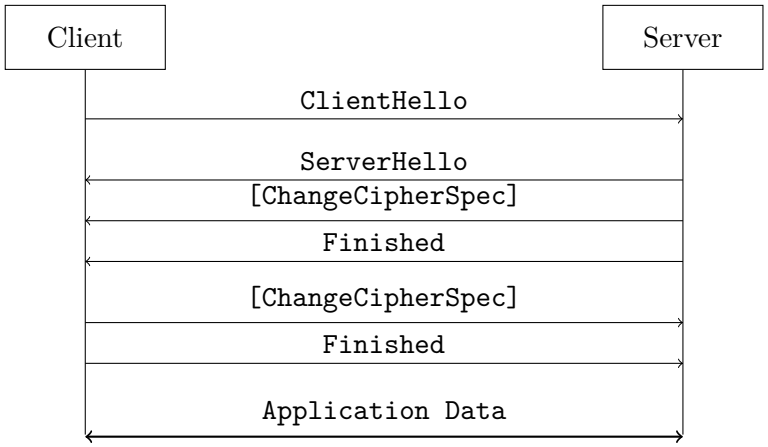

Figure 2.2: Simplified TLS handshake

Let's have a closer look at the various messsages that are exchanged during the TLS handshake:

`HelloRequest`

This message may be sent by the server at any time and tells the client to begin a new session negotiation. Client should respond to this message with a `ClientHello`. This message is not often used, but it can be useful in some cases, e.g. forcing a session renegotiation for a TLS sessions which are active for a longer period of time.

`ClientHello`

The `ClientHello` message is usually the first message in the TLS handshake, sent by the client, which initiates the session negotiation. The message itself contains the latest TLS version supported by the client, the client's random value, a session ID (which can be empty, if the client wishes to negotiate a new session), a list of supported cipher suites, a list of compression methods and optional extensions.

`ServerHello`

The `ServerHello` is the server's response to the client's `ClientHello` message. The structure of this message is quite similar to the `ClientHello` – instead of the lists of cipher suites and compression methods, the server specifies a single cipher suite and a single compression method. These values are chosen from the client's `ClientHello` message and will be used for the session. The complete message contains the TLS version chosen by the server, the server's random value, the length of the session ID and the session ID, the cipher suite, the compression method and the optional extensions.

`Certificate`

If the agreed-upon key exchange method uses certificates for authentication, the server sends a `Certificate` message containing a certificate chain. This certificate chain can be then used by the client, to verify the server's identity. The same message is used when a client authentication is required (as a response to a `CertificateRequest` message, see

below). All exchanged certificates are of X.509 v3 type [2], if not stated otherwise during the negotiaton.

`ServerKeyExchange`

In some cases, the `Certificate` message does not contain enough data to allow the client to exchange a premaster secret. In this situation, the server sends a `ServerKeyExchange` message with necessary cryptographic information, which allows such exchange and allows the client to complete a key exchange.

When RSA (or Diffie-Hellman with fixed parameters) is used, the client can retrieve the public key (or the server's Diffie-Hellman parameters) from the server certificate. In these cases the `Certificate` message is enough to complete a key exchange. But, for example, in case of ephemeral Diffie-Hellman, the client needs some additional information — Diffie-Hellman parameters — which must be delivered in a `ServerKeyExchange` message.

`CertificateRequest`

When the server wants to authenticate the client, it sends a `CertificateRequest` message to the client. This message tells the client, which certificates are accepted by the server, and also asks the client to send its certificate to the server. Only a non-anonymous server can send a `CertificateRequest` – that means a server, which authenticates itself using the `Certificate` message.

`ServerHelloDone`

This message is sent by the server and indicates the end of the section of messages initiated by the `ServerHello` message. After sending this message, the server will wait for a client response.

`ClientKeyExchange`

The `ClientKeyExchange` message is sent by the client and it provides the server with the client-side keying material, which is used to generate the premaster secret.

`CertificateVerify`

If the client provided a certificate that has signing capability, then it must prove that it holds the corresponding private key for that certificate. In this case, the client sends a digitally signed `CertificateVerify` message to the server. This allows the server to verify the client certificate using the client's public key and authenticate the client.

`Finished`

The `Finished` message is always sent immediately after a `ChangeCipherSpec` message 2.2.3 to verify that the key exchange and authentication processes were successful. As the message follows the `ChangeCipherSpec` message, it is the first message protected by the newly negotiated algorithms and keys.

2.2.3 TLS Change Cipher Spec Protocol

The TLS Change Cipher Spec Protocol is used to signal transitions in ciphering strategies. The protocol itself consists of a single compressed and encrypted message – `ChangeCipherSpec`. The encryption and compression methods correspond to the current (not the pending) cipher spec.

After receiving this message, the receiver instructs its record layer to immediately copy the read pending state into the read current state. Similarly, immediately after sending this messsage, the sender instructs its record layer to copy the write pending state into the write current state. All subsequent messages sent by the sender are then protected under the newly negotiated cipher spec.

2.2.4 TLS Alert Protocol

To exchange alert messages between peers, like warnings and errors, the TLS Alert Protocol is used. Each alert message has its severity (warning, fatal) and a description of the alert. All messages with an alert level of fatal result in the immediate termination of the connection.

The alert's description field contains an identificator of the alert. These descriptions can be split into two categories – closure alerts and error alerts. The former one contains only one alert – `close_notify`. This message can be send by either party and notifies the recipient that the sender will not send any more messages. Any data received after this message must be ignored. The knowledge of the fact, that the connection is ending, is crucial to avoid truncation attacks. The second category contains a number of error alerts used for various purposed during the TLS session. All TLS alert messages are summarized in Table A.1.

2.2.5 TLS Application Data Protocol

The TLS Application Data Protocol takes the arbitrary data from the application layer and feeds it into the TLS Record Protocol for fragmentation, compression and encryption. The resulting TLS records are then sent to the recipient.

2.3 Cipher Spec and Cipher Suite

In previous sections we mentioned terms *cipher spec* and *ciper suite*. Let's have a closer look at their meanings.

A *cipher spec* refers to a pair of algorithms that are used cryptographically protect data. Such pair consists of a message authentication algorithm (*MAC*) and a data encryption algorithm. If we add a key exchange algorithm to a cipher spec, we get a *cipher suite*.

For example, *TLS_DHE_RSA_AES_256_CBC_SHA1* refers to a TLS cipher suite which uses ephemeral Diffie-Hellman with RSA for a key exchange, 256-bit AES in CBC [8] mode for encryption, and SHA-1 for message authentication. **[[List of cipher suites?]]**

[8] Cipher Block Chaining

2.4 TLS Extensions

As mentioned in section 2.2.2, the `ClientHello` and `ServerHello` messages contain an optional field for extensions. These extensions can be used to add functionality to TLS.

When a client wants to use an extension it sends its name in the `ClientHello` message. If the extension is supported by the server, it will be included in the responding `ServerHello` message. However, if the `ServerHello` message contains an extension, which was not sent by the client, the connection must be aborted with an `unsupported_extension` fatal alert (A.1). [7] [8]

Describing all currently implemented extensions [3] is way beyond scope of this thesis. Thus, in the following paragraphs, we will discuss only those extensions, which are currently tested by the implementation part of this thesis. Nevertheless, support for other extensions is highly probable in the near future.

2.4.1 Session Tickets

If a client wanted to resume an existing session, it would have to send a session ID in its `ClientHello` message (field `session_id`) and the server would have to check its cache for a match (see section 2.2.2). This may cause problems on systems with a large amount of requests from different users or on systems with little memory. For such cases there is a `SessionTicket` TLS extension which uses client-side caching. [5]

In the initial handshake, where the client does not possess a ticket for an existing session, it includes an empty `SessionTicket` TLS extension in its `ClientHello` message. The server responds with an empty `SessionTicket` extension to indicate that it will send a new session ticket using the `NewSessionTicket` message. This ticket contains the current session state (such as ciphersuite and master secret) and is cryptographically protected by a key, which is known only to the server.

When the client wishes to resume the session, it includes the cached ticket in the `SessionTicket` extension of its `ClientHello` message. The server then decrypts and verifies the contents of the ticket and resumes the session according to the decrypted parameters. If the server cannot or does not want to use the state from the ticket, then it can initiate a full handshake with the client.

2.4.2 Signature Algorithms

TLS 1.2 defines ([11], section 7.4.1.4.1) an extension `supported_signature_algorithms`, which allows a client to tell the server which hash and signature algorithm combinations it supports. Even though internally the supported algorithms are split into two lists (none, MD5, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 for hash algorithms and anonymous, RSA, DSA and ECSDA for signature algorithms), the algorithms sent in the extension are always listed in pairs, as not all combinations may be accepted by an implementation.

The peers exchange this information through the `ClientHello` and `ServerHello` messages including the `supported_signature_algorithms` extension. The client sends a list of supported algorithms and the server responds with its choice, that is going to be used for any subsequent signature generation and verification.

Chapter 3

Continuous Integration System

[[(overview, jenkins, travis, semaphore, beaker, openstack, ...)]] [[current state - github, travis, docker, ...]]

3.1 GitHub & Webhooks

3.2 Jenkins

3.2.1 Plugins(?)

3.3 Beaker

3.4 OpenStack

[[+components]]

3.5 Travis

3.6 Docker

3.7 Current State

Chapter 4

SSL/TLS Testing

Before the testing itself, we have to know what should be tested, how it should be tested and where, or to be more precise, which environments it should be tested in.

4.1 Tested Libraries

To prevent each project implementing the SSL/TLS on its own and introducing (in many situations) dangerous issues, several libraries were created and can be used by any project, which needs a SSL/TLS support. The most popular ones are OpenSSL [1], NSS [2] and GnuTLS [3]. Although, there are other SSL/TLS libraries (e.g. LibreSSL [4] or BoringSSL [5]), this thesis aims only on these three. However, a future expansion to support other libraries is not impossible.

Even though these libraries are separate projects, an user must be able to communicate with every client, which supports the particular protocol and ciphersute, no matter which implementation they use. Testing of this functionality — *interoperability* — is the main goal of this thesis.

For the testing itself we need at least two applications - a client and a server. One option would be writing these applications using the public API [6] of each library, which is not error prone and would require a maintenance of such applications. Thankfully, each of the tested libraries provides a set of utilities, among which we can found a simple client and server application with dozens of settings and options. These applications are then used in various scenarios to ensure, that given valid combination of settings works for both client and a server using different libraries.

4.1.1 OpenSSL

OpenSSL is an open source library maintained by The OpenSSL Project, which provides a toolkit for TLS and SSL protocols, along with other general-purpose cryptographic functions.

[1] https://www.openssl.org/
[2] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
[3] https://www.gnutls.org/
[4] https://www.libressl.org/
[5] https://boringssl.googlesource.com/boringssl/
[6] Application Programming Interface

This library provides a single powerful utility called *openssl*. This utility has dozens of subcommands with various SSL/TLS related functionality, where the most important ones are:

ciphers information about supported ciphersuites

dsa, rsa, ec DSA/RSA/EC key management

s_client a simple SSL/TLS client

s_server a simple SSL/TLS server

x509 X.509 certificate data management

Thanks to its functionality-rich interface, OpenSSL is used for certificate generation and management for all libraries in the testing process.

4.1.2 NSS

Network Security Services (NSS) is a set of open source libraries providing support for SSL and TLS protocols, S/MIME [7] and other optional features, like server-side TLS/SSL acceleration or client-side hardware smart cards support.

Compared to OpenSSL, which has a single utility for everything, NSS does the exact opposite - each feature or tool has its own utility. For our testing purposes, we will need the following ones:

certutil certificate and key management for NSS databases

listsuites information about supported ciphersuites

selfserv a simple SSL/TLS server

strsclnt a simple SSL/TLS client for performance testing

tstclnt a simple SSL/TLS client

This library differs from the other two in the way how it handles server and client certificates. These certificates cannot be passed directly as arguments to the utility, but must be imported to a NSS database which is then passed as an argument to the utility.

4.1.3 GnuTLS

GnuTLS is a secure communications library which implements SSL, TLS and DTLS [8] protocols.

Like the the libraries above, GnuTLS includes several utilities for library testing - GnuTLS client *gnutls-cli* and GnuTLS server *gnutls-serv*. Both utilities support a parameter **-l**, which lists all necessary information about supported ciphersuites.

[7] Secure/Multipurpose Internet Mail Extensions
[8] Datagram Transport Layer Security

4.2 Tested Environments

For the purposes of this thesis, an environment consists of an operating system (e.g. CentOS [9], Fedora [10]) and its version (e.g. 7, 25). Each of these environments must be tested separately, as it contains a different set of library versions and policies, which affect the SSL/TLS communication.

This thesis covers SSL/TLS libraries on CentOS and Fedora, as the tests used and extended by this thesis were originaly created for RHEL [11].

4.3 Test Format

[[test plan(!!!)]]

4.3.1 Beakerlib

4.3.2 Test relevancy

4.4 Testing Process

[[PR/commit trigger -> test-setup.sh -> docker -> test-runner.sh ...]]

4.5 Outstanding Issues(?)

[[FIPS]]

[9] https://www.centos.org/
[10] https://getfedora.org/
[11] Red Hat Enterprise Linux - https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

Chapter 5

Testing Results

[[filed bugs/CVEs, split upstream and downstream]]

Chapter 6

Conclusion

[[summarize findings, current CI state, future work (if any), ...]]

Bibliography

[1] A. Freier, P. Karlton, P. Kocher: The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101. RFC Editor. August 2011.
Retrieved from: http://www.rfc-editor.org/rfc/rfc6101.txt

[2] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280. RFC Editor. May 2008.
Retrieved from: http://www.rfc-editor.org/rfc/rfc5280.txt

[3] D. Eastlake: Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066. RFC Editor. January 2011.
Retrieved from: http://www.rfc-editor.org/rfc/rfc6066.txt

[4] Duong, T.; Rizzo, J.: Here Come The XOR Ninjas. 2011.

[5] J. Salowey, H. Zhou, P. Eronen, H. Tschofenig: Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077. RFC Editor. January 2008.
Retrieved from: http://www.rfc-editor.org/rfc/rfc5077.txt

[6] R. Barnes, M. Thomson, A. Pironti, A. Langley: Deprecating Secure Sockets Layer Version 3.0. RFC 7568. RFC Editor. June 2015.
Retrieved from: http://www.rfc-editor.org/rfc/rfc7568.txt

[7] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, T. Wright: Transport Layer Security (TLS) Extensions. RFC 3546. RFC Editor. June 2003.
Retrieved from: http://www.rfc-editor.org/rfc/rfc3546.txt

[8] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, T. Wright: Transport Layer Security (TLS) Extensions. RFC 4366. RFC Editor. April 2006.
Retrieved from: http://www.rfc-editor.org/rfc/rfc4366.txt

[9] T. Dierks, C. Allen: The TLS Protocol Version 1.0. RFC 2246. RFC Editor. January 1999.
Retrieved from: http://www.rfc-editor.org/rfc/rfc2246.txt

[10] T. Dierks, E. Rescorla: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4643. RFC Editor. April 2006.
Retrieved from: http://www.rfc-editor.org/rfc/rfc4346.txt

[11] T. Dierks, E. Rescorla: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. RFC Editor. August 2008.
Retrieved from: http://www.rfc-editor.org/rfc/rfc5246.txt

Appendices

Appendix A

TLS Alerts

Table A.1: TLS Alerts

Alert	ID	Description
`close_notify`	0	The sender notifies the recipient that it will not send any more messages on this connection.
`unexpected_message`	10	An inappropriate message was received. This alert is always fatal.
`bad_record_mac`	20	The sender received a record with an incorrect MAC. This alert is always fatal.
`decryption_failed_RESERVED`	21	Used in some earlier versions of TLS, must not be sent by compliant implementations.
`record_overflow`	22	A `TLSCiphertext` record was received that had a length more than $2^{14} + 2048$ bytes or a record decrypted to a `TLSCompressed` record with more than $2^{14} + 1024$ bytes. This alert is always fatal.
`decompression_failure`	30	The decompression function received improper input. This alert is always fatal.
`handshake_failure`	40	The sender was unable to negotiate an acceptable set of security parameters given the options available. This alert is always fatal.
`no_certificate_RESERVED`	41	This alert was used in SSLv3 but it no longer used in any TLS version.
`bad_certificate`	42	The sender notifies the recipient that the provided certificate is corrupt.
`unsupported_certificate`	43	The sender notifies the recipient that the provided certificate is of an unsupported type.
`certificate_revoked`	44	The sender notifies the recipient that the provided certificate was revoked by the issuing authority.
`certificate_expired`	45	The sender notifies the recipient that the provided certificate has expired or is no longer valid.
`certificate_unknown`	46	The sender notifies the recipient that some unspecified issue occured during the certificate processing, rendering it unacceptable.

Table A.1: TLS Alerts

Alert	ID	Description
`illegal_parameter`	47	A field in the handshake was out of range or inconsistent with other fields. This alert is always fatal.
`unknown_ca`	48	The received certificate could not be validated, because the CA certificate could not be located or could not be matched with a known, trusted CA. This alert is always fatal.
`access_denied`	49	A valid certificate was received, but when access control was applied, the sender decided not to proceed with negotiation. This alert is always fatal.
`decode_error`	50	The received message could not be decoded because some field was out of the specified range or length of the message was incorrect. This alert is always fatal.
`decrypt_error`	51	A handshake cryptographic operation failed. This alert is always fatal.
`export_restriction_RESERVED`	60	Used in some earlier versions of TLS, must not be sent by compliant implementations.
`protocol_version`	70	The protocol version the client has attempted to negotiate is recognized but not supported. This alert is always fatal.
`insufficient_security`	71	The server requires more secure ciphers than those supported by the client. This alert is always fatal.
`internal_error`	80	An internal error occured, unrelated to the peer or corectness of the protocol. This alert is always fatal.
`user_canceled`	90	This handshake is being canceled for some reason unrelated to a protocol failure. This alert should be followed by a `close_notify`.
`no_renegotiation`	100	The peer should respond with this alert when renegotiation is not appropriate regarding the current connection state. This alert is always a warning.
`unsupported_extension`	110	Sent by the client when the received `ServerHello` message contains an extension not sent by the client in its `ClientHello` message. This alert is always fatal.