BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

TLS/SSL INTEROPERABILITY ACROSS SYSTEMS

TLS/SSL INTEROPERABILITA MEZI SYSTÉMY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Peter Leitmann

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. Mgr. Pavel Šeda, Ph.D.

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Information Security**

Department of Telecommunications

Student: Peter Leitmann ***ID:*** 230110

Year of study: 3 ***Academic year:*** 2022/23

TITLE OF THESIS:

TLS/SSL Interoperability Across Systems

INSTRUCTION:

The goal of the bachelor thesis is to improve quality by executing TLS interoperability tests earlier in the development process. Get familiar with existing TLS interoperability tests used in Red Hat Enterprise Linux (RHEL). The student will also learn about the Continuous Integration and Continuous Delivery/Deployment (CI/CD) model. Further, the student will explore the CI systems used in upstream TLS libraries: OpenSSL, GnuTLS, and NSS. Next, the student will discover CI systems in upstream Linux distributions related to RHEL - CentOS Stream, and Fedora. The output of the bachelor thesis will include the integration of TLS interoperability tests into at least three of these projects (OpenSSL, GnuTLS, NSS, CentOS Stream, and Fedora).

RECOMMENDED LITERATURE:

[1] GitLab CI/CD [online]. [cit. 2022-09-08]. Dostupné z: https://docs.gitlab.com/ee/ci/

[2] Test Management Tool [online]. [cit. 2022-09-08]. Dostupné z: https://tmt.readthedocs.io/en/stable/

Date of project specification: 6.2.2023 ***Deadline for submission:*** 26.5.2023

Supervisor: Ing. Mgr. Pavel Šeda, Ph.D.

Consultant: Ing. Stanislav Židek

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

ABSTRACT

The main aim of this thesis is to add new interoperability tests to selected upstream projects. The requirement for such a thing originates in a need of developers to develop software more efficiently. This thesis describes widely used security protocols – Secure Sockets Layer (SSL) and Transport Layer Security (TLS). Subsequently, it focuses on the process how software should be developed using Continuous Integration and Continuous Deployment/Delivery (CI/CD), which consists of not only creating code but also of testing it in an efficient way. Particularly, Software Development Life Cycle and Software Testing Life Cycle are explained. In the practical part, the process of choosing suitable tests can be seen, a configuration of various CI systems is described, and testing of the developed software. The test environment is the operating system Fedora, and tests rely on an open-source tool called Test Management Tool.

KEYWORDS

CI, CI/CD, GnuTLS, interoperability, NSS, OpenSSL, Red Hat, testing, TMT, QE

ABSTRAKT

Hlavným cieľom tejto bakalárskej práce je pridanie nových testov interoperability do vybraných open-source projektov. Dôvod pre toto pramení v dôraze na zefektívňovanie práce vývojových tímov. Táto práca hlavne popisuje široko používané bezpečnostné protokoly – Secure Sockets Layer (SSL) and Transport Layer Security (TLS). Ďalej sa zameriava na popis postupu, ako by mal byť softvér vyvíjaný s použitím CI/CD (priebežného integrovania a priebežného nasadzovania), čo pozostáva nie len z vytvárania kódu ale aj jeho testovania. Preto je v práci vysvetlený známy cyklus vývoja SDLC (Software Development Life Cycle) a aj cyklus testovania STLC (Software Testing Life Cycle). V praktickej časti je popísaný proces vyberania vhodných testov, konfigurácia rôznych CI systémov a následné testovanie vyvíjaného softvéru. Testovacie prostredie je výhradne operačný systém Fedora a testy sú závislé na open-source nástroji Test Management Tool.

KĽÚČOVÉ SLOVÁ

CI, CI/CD, GnuTLS, interoperabilita, NSS, OpenSSL, Red Hat, testovanie, TMT, QE

ROZŠÍRENÝ ABSTRAKT

Bezpečnosť na internete sa stala neodmysliteľnou súčasťou všetkých aplikácií. Pre dosiahnutie hlavných cieľov bezpečnej komunikácie na internete boli takmer pred dvomi dekádami predstavené prvé bezpečnostné protokoly. Hlavnými protagonistami tejto problematiky bol protokol SSL (Secure Sockets Layer) a neskôr jeho následníci. SSL významne prispel k vzniku TLS (Transport Layer Security), ktorý sa v určitých verziách (najmä verzia 2 a 3) momentálne používa k zabezpečeniu. Teoretickému pozadiu a rozboru funkcionality uvedených protokolov sa venuje prvá kapitola.

Implementáciu tohto protokolu do aplikácií sprostredkovávajú najčastešie popredné bezpečnostné knižnice, ako napr. OpenSSL, GnuTLS a NSS. Avšak kvôli skutočnosti, že knižníc je viac, je taktiež možné, že počas ich vývoja medzi nimi vzniknú inkonzistencie, ktoré spôsobia vzájomnú, či už menšiu alebo väčšiu, nekompatibilitu.

Práve z tohoto dôvodu sa využívajú takzvané testy interoperability, ktoré je dôležité počas vývoja vykonávať. Kvôli tejto skutočnosti vznikla táto práca, a to za účelom rozšírenia testov interoperability tam, kde je to potrebné. Aj do projektov, kde už sa niekoľko testov nachádzalo, ale aj do projektov, kde takéto testy doposial neboli v skoršej vývojovej fáze.

Keďže sa jedná o vývoj a testovanie softvéru, problematika je úzko spojená s technológiami CI (Continuous Integration – Priebežná Integrácia). Aplikovanie technológie CI je pri vývoji väčšími skupinami ľudí výhodné. Jej použitím je vývoj omnoho lepšie kontrolovaný a v prípade ideálneho nasadenia objavuje chyby v zdrojových kódoch skôr, ako má nechcený dopad na výsledný softvér.

To, ako je v dnešnej dobe najčastejšie softvér vyvíjaný, je možné vidieť v druhej kapitole. V nej sa nachádzajú základné procesy vývoja a testovania (Software Testing Life Cycle a Software Development Life Cycle), ktoré často aplikujú v praxi.

Ďalej sa je možné oboznámiť aj s ďalšími nástrojmi ako sú napr. TMT, BeakerLib, Podman, ktoré boli využité počas priebehu praktickej implementácie práce.

V praktickej časti sa je možné stretnúť s niekoľkými CI systémami, a to GitLab CI/CD, Zuul a GitHub Actions. GitLab CI/CD je vyžívaný v projekte GnuTLS, ktorý je umiestnený priamo v GitLab repozitári. Zuul CI je vo veľkej miere používaný v projektoch, ktoré vyvíja spoločnosť Red Hat. Aj preto pri prispievaní do projektov CentOS Stream a Fedora bol priamo využívaný tento systém. Pri prispievaní do upstream projektu OpenSSL sa využíva priamo systém GitHub Actions. Tam je možné vidieť náležitosti vytvorenia testovacieho prostredia a exekúciu testov.

Výstup tejto práce je možné sledovať v poslednej kapitole, kde je popísané úspešné rozšírenie množiny testov do projektov GnuTLS, Fedora a CentOS Stream. V prípade upstream projektu OpenSSL je situácia mierne komplikovanejšia, pretože sa jedná o externý projekt, v ktorom je nevyhnutná diskusia ohľadom vyhovujúceho nasadenia týchto testov, preto sa na dokončení práce stále pracuje.

LEITMANN, Peter. *TLS/SSL interoperability across systems*. Brno: Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 49 p. Bachelor's Thesis. Advised by Ing. Mgr. Pavel Šeda, Ph.D.

Author's Declaration

Author:	Peter Leitmann
Author's ID:	230110
Paper type:	Bachelor's Thesis
Academic year:	2022/23
Topic:	TLS/SSL interoperability across systems

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to express my sincere thanks to Ing. Mgr. Pavel Šeda, Ph.D. and Ing. Stanislav Židek, for their insightful remarks, expertise, patience, and guidance during the writing of this thesis. I would also like to convey my heartfelt gratitude to my family and close friends for their unwavering support throughout this journey.

Contents

List of Figures

Listings

Introduction

This thesis has more of an implementation character than a creative one. The main objective of this thesis is to enhance the quality of the development process of cryptographic libraries. This is done by adding interoperability tests to earlier development stages. By doing this, it is possible to notice problems sooner and therefore can save costs and time.

The subject of testing is Transport Layer Security (TLS) implemented in multiple Linux distributions developed by Red Hat, Inc. It is important to state that TLS can be implemented using various libraries like GnuTLS, OpenSSL, or NSS (Network Security Services). Considering this, compatibility between all these libraries needs to be assured. The ability of these libraries to intercommunicate is called "interoperability". The most important aspects considered are the negotiation of a new connection and the resumption of a previous connection.

In the initial part of the thesis, the main focus is on the GnuTLS library. Later on, other projects are described – CentOS Stream and Fedora (Rawhide[1]). In the last part, the upstream OpenSSL project is approached.

The second part of this thesis is to work with Continuous Integration and Continuous Deployment (CI/CD) systems. CI systems play an important role in software development as they provide useful feedback in the early stages of development. It is vital to know, whether the changes made do not cause regressions, or need further adjustments. Thanks to this, code can be developed quicker and with higher quality.

In the case of the other projects, they do not rely on GitLab's CI. CentOS and Fedora use a gating system called Zuul. There are multiple advantages of this system since there exists a good integration of the Test Management Tool. In the last project, upstream OpenSSL, the situation is different. The technology used in this project is GitHub Actions and needs a similar way of configuration to the upstream GnuTLS project.

The practical output of this thesis is the addition of interoperability tests into the GnuTLS upstream library. Furthermore, the addition of these tests to GnuTLS, OpenSSL and NSS libraries for Red Hat's CentOS Stream and Fedora. Lastly, the addition of tests to the upstream OpenSSL library has begun but to this day, it has not been completed. A new group of tests has been implemented after inspecting the relevancy and correctness of the tests.

The first chapter describes the foundations of SSL (Secure Socket Layer) and TLS protocols and compares the last two releases of TLS. Afterwards, product development and the importance of CI/CD are explained. The last chapter covers the practical part of the thesis.

[1]This is the main development branch of Fedora operating system.

1 Security layer

1.1 Secure Sockets Layer/Transport Layer Security

The world has massively evolved in terms of information technologies and the interconnection of billions of machines and devices has become today's reality. With the rise of the Internet of Things in the last century, and the rapid growth of users connected to the Internet, a natural need for security and especially the privacy of this platform has become the number one priority for many. In order to provide the needed security, there has to exist a channel, where parties can exchange messages securely [1].

1.1.1 Secure Sockets Layer overview

In this regard, back then, Secure Sockets Layer was the leading candidate to revolutionize the Internet. The history of SSL dates back to February 1995, to its first public release, SSL 2.0. Version 1.0 was never publicly released due to its severe security vulnerabilities. However, not even the second version was in service for long. As no deep expertise had been conducted by security experts during the development, and consequently security deficiencies (e.g., use of MD5[1] based MAC[2]) had been found, developers replaced version 2.0 with the new 3.0 version in 1995. Unfortunately, the new protocol was not successful in fixing all flaws. In spite of the same name, the new version was much better and sets the foundational stone for its successor which is broadly known today as TLS (Transport Layer Security) [1, 2].

This thesis will not conduct any further discussions of SSL as its latest version 3.0 was deprecated in 2015 by IETF[3] [3].

1.1.2 Transport Layer Security overview

The main goals of SSL and TLS are to provide confidentiality (data are not visible to third parties), data integrity (data are not changed during transfer) and authentication by exchanging digital certificates [4]. Because TLS resulted from an ownership takeover[4] [5], the first release 1.0 was almost identical to the SSL 3.0, only with a few minor changes. More changes were brought with the release of TLS extensions in 2003 and TLS 1.1 in 2006 fixed some security flaws of the protocol [1].

[1] Message Digest Algorithm 5

[2] Message Authentication Code

[3] Internet Engineering Task Force

[4] Netscape and Microsoft were developing the protocol. To standardize the protocol, IETF took over the development process.

It is undoubtful that a significant success of this protocol happened with the release of TLS 1.2 in 2008. Evidence of this can be seen in many places of today's internet world where TLS 1.2 is still utilized as one of the main protocols to secure the communication [1].

One of the biggest changes from TLS 1.1 are:

- Possible authenticated encryption.
- TLS extensions incorporated into the main protocol.
- Pseudorandom functions (PRF) can now be specified by cipher suites [1].

Later, a newer version (TLS 1.3) was released in April 2014. Exactly 28 drafts were needed to completely define the new TLS 1.3 protocol. The final release was in August 2018 [6].

1.1.3 TLS 1.2

The main goal of TLS protocol is to establish a secure connection which cannot be compromised by attackers. The real challenge is to produce a protocol that can be widely used and will not have to be soon replaced by other protocol, which can lead to exposing communication to new possible vulnerabilities [1, 7]. The TLS 1.2 protocol was undoubtedly a great success. It got integrated into billions of systems connected to the internet. However, the protocol is not ideal [8]. The protocol targets these goals:

- Cryptographic security – A secure connection between entities.
- Interoperability (which is the main topic of this thesis) - ensuring systems that share no knowledge of each other's code, are able to agree on e.g., cipher suite and other parameters and communicate [1, 7].
- Extensibility – this means, the way TLS works is not rigid and it is adjustable to the current needs of a particular application. This aims towards a framework in which for example new encryption methods can be set up [1, 7].
- Relative efficiency – cryptographic operations can be extensively CPU (Central Processing Unit) demanding. Therefore, TLS provides a session caching mechanism to reduce the number of connections, which have to be created from scratch [1, 7].

The protocol has two layers – TLS Record Protocol and TLS Handshake Protocol. It is usually built on top of a reliable transport protocol e.g., TCP[5] (see Figure 1.1).

[5]Transmission Control Protocol

1.1.4 Record Protocol

The main functions of the protocol are to transport and possibly encrypt the messages. The record layer fragments, assigns a Message Authentication Code (MAC) and sends blocks of data - `records`. The compression option was found to be dangerous, therefore it became restricted[6] [9]. `Records` consist of a header, which informs about the type of content that is being carried and a data structure. Other attributes obtained in the header are the protocol version and length. According to the character of the data, there are three data structures that can be created – `TLSPlaintext`, `TLSCompressed` and `TLSCiphertext`. This layer is responsible for the extensibility of the protocol with new subprotocols thus prolonging the lifetime of the TLS protocol itself [1, 7, 10].

1.1.5 Handshake Protocol

This protocol is the most crucial part of TLS. In order to establish a new TLS connection, the endpoints need to resolve possible compatibility obstacles. This is provided by the Handshake protocol. It is built on top of the Record protocol (see Figure 1.1) and ensures that all security parameters of a session are set. The handshake messages are encapsulated in the `records` and transmitted further [1, 7, 10].

The protocol itself provides a way of creating a secure session, however, higher layers cannot merely rely on the fact, that negotiation of the handshake is successful. Concerning the fact that the connection could be compromised by e.g., a "Man in the Middle" attack, the essential thing is that the higher layers must check, which parameters and properties were negotiated and if that is the security standard allowed to be used [1, 7, 10].

TLS Handshake protocol operates in the following steps:

- Negotiation of a protocol version.
- Exchange of information about available algorithms and picking one.
- Authentication – in most cases, the server's side authenticates, optionally, the client can do so too.
- Negotiation of a session key.

Full handshake is shown in Figure 1.2. Messages marked by "*" are not always sent. The "[ ]" indicate that this is not a TLS handshake message [7].

[6] CRIME (CVE-2012-4929) vulnerability.

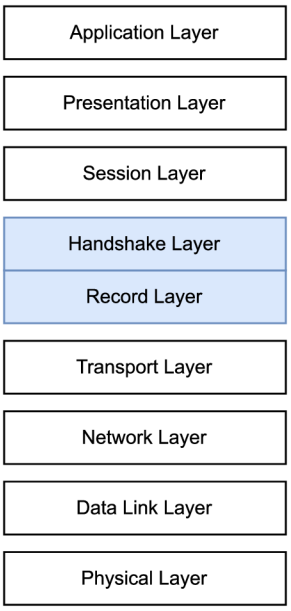

Fig. 1.1: ISO/OSI model – TLS

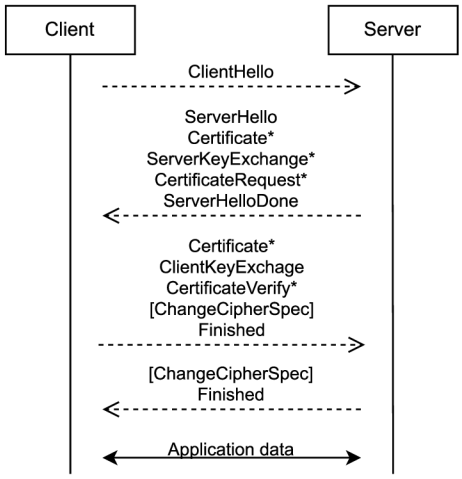

Fig. 1.2: TLS 1.2 full handshake

Hello Request

The `Hello Request` is a message used by a server during a session. It notifies the client's side that it should renegotiate a new session. However, this message should not be sent immediately after establishing a connection. In order to continue the already started negotiation successfully, the client needs to respond with a `ClientHello` message. If not done so, the server can close the session with a fatal alert. This message is neither allowed to be contained in the message hashes used in the `Finished` messages [1, 7].

Client Hello

This message is sent by the client. It can be sent in various moments. When connecting to a server, this should be the first message to be sent. In case a server-client's session is already underway, the client can trigger a renegotiation of the session parameters with this message. It is possible to reuse some parameters from the current session and renegotiate just the particular ones. The `ClientHello` message contains the client's cipher suites ordered by preference. In detail, the cipher suite specifies a key exchange algorithm, a bulk encryption algorithm, MAC and PRF. When this message is received, the server picks the most favourable compatible algorithm and replies with a `ServerHello` response. If there are no compatible ways of proceeding further with the handshake, the server replies with a fatal error and the connection becomes interrupted. This message also contains information about the extensions wished to be used [1, 7, 11].

Server Hello

The `ServerHello` message is used to provide information on how the client's `Hello` message is resolved. The server either agrees to the proposed attributes by the client or the connection is terminated with a fatal error. [1, 7, 11].

Certificate

This message is an optional one and follows the `ServerHello`. It occurs if the server and client agree on authentication by certificates. The certificates are by default a X.509v3 type unless the parties agree differently [1, 7, 11].

Server Key Exchange Message

This message is also optional and is sent only in specific scenarios. It is used to provide additional information to the client if the `ServerHello` is not sufficient enough. This message can only happen in combination with specific key exchange methods: DHE_DSS, DHE_RSA, and DH_anon. With this additional information, the client is able to generate a pre-master secret of the handshake [1, 7, 12].

Certificate Request

A server that authenticates itself with a certificate can ask for a certificate to authenticate a client. In this case, the client sends a specifically signed certificate and proves his identity. A fatal handshake error is raised if the client is sent a request for authentication by an anonymous server [1, 7, 11].

Server Hello Done

This message is sent to a client after the `ServerHello` message sequence is finished. After that, the server starts to wait for a client's response [1, 7].

Client Certificate

Sending this message is relevant only if the server requires a certificate. It is sent when the server finishes its part of authentication and provides the client with all requested information. The client is now obligated to send a certificate signed by a trusted authority. If there is no appropriate certificate, the client sends a blank message, otherwise, the server can abort the handshake or handle the client as unauthenticated. In case the message is complete, it is used for verification of `CertificateVerify` message or for creating the pre-master secret. The certificate must meet similar requirements as the server's one – X.509v3 (if not negotiated differently) [7, 1].

Client Key Exchange Message

This message is mandatory. It transmits the pre-master secret, the ECDH parameters. There are a few ways how this can happen. Either the client sends an RSA-encrypted secret or provides Diffie–Helman parameters which are to be used to calculate the secret on each side [1, 7, 12].

Certificate Verify

This message is sent in response to the `ClientCertificate` message. It contains all the messages (their signatures) exchanged until this point. This means that the server and the client need to prepare in advance for this step and save the handshake messages continuously. With this message, the server verifies that the client is the owner of the private key of the certificate [1, 7].

Finished

This message is always sent right after the `ChangeCipherSpec` message (see Section 1.1.6). This is the first encrypted message that is sent. It provides a verification that the handshake was successful. It is followed by a `ChangeCipherSpec` response and another `Finished` message from the server's side. When both sides verify the content of the messages, the application data can now be transferred [1, 7, 11].

All the messages exchanged between a client and a server must be in the prescribed order. Otherwise, the connection will fail on either side. The only exception is the `HelloRequest` message, which is handled differently [7].

1.1.6 Change Cipher Spec Protocol

This is a subprotocol used to inform the communicating sides about the ciphering mechanisms used with the following messages. The way the protocol operates is by sending a single message to the other side. The message is encrypted and compressed under the current state [1, 7, 12].

1.1.7 Alert Protocol

The alert protocol is used as a way of handling errors occurring in the ongoing session. The protocol message is made of two parts – alert level and alert description. The connection is strongly dependent on these messages. If an alert message is transmitted with "fatal" severity, the connection is immediately terminated and the rest of the connections of that session are degraded into a state, where the ongoing

messages can continue to be transmitted, however, this session can no longer be resumed under any circumstances [1, 7].

1.2 TLS 1.3

This section compares the previously described version (see Section 1.1.3) with the latest release of the TLS protocol 1.3. The changes in some areas are significant, and overall, the protocol has become faster and more secure. Although the protocol is rather new, in the August of 2021, it was measured that 63% of the top million web servers preferred the use of TLS 1.3 [13].

Unfortunately, TLS 1.3 is not directly backwards compatible with TLS 1.2. Therefore, all TLS protocols have a versioning system which resolves this problem [8].

1.2.1 Major differences from TLS 1.2

The cipher suites

A basic, yet important change that happened in the new release, is the modification of available cipher suites. The number decreased from 37 to 5 supported cipher suites. The legacy algorithms could no longer stay and had to be removed. Algorithms that are not inevitably unsafe, but their wrong implementation could expose communication to danger, were also excluded. The cipher suites have now a common feature – forward secrecy. The main goal of forward secrecy is that a new session key is created for each connection. When following this rule, only a limited amount of data becomes vulnerable when a private key is captured or the algorithm is broken by an attacker [14].

Particularly concerning was the use of the RSA key exchange. It was used in the initial part of the handshake to set up symmetric keys to encrypt the application data of the connection. This could be a good approach, however, it has some flaws. The main problem is the length of validity of the symmetric keys (4-5 years). If the keys get captured by the attacker, which is possible, all the connections made in the past with that one specific key become vulnerable. This is solved by the switch, in the new release, to a new key exchange algorithm – ephemeral Diffie-Helman. After all, for backward compatibility purposes, there is an option to use RSA-PSS (RSA-Probabilistic Signature Scheme) for signatures in certificates. The PKCS #1 (Public-Key Cryptography Standards 1) cannot be used for this purpose anymore [15, 16].

Another security improvement that has been made is that all present algorithms are now combined with Authenticated Encryption with Associated Data (AEAD). This type of encryption guarantees data to be confidential and authentic [15, 16].

The handshake

The full handshake is far from perfect, reflecting on the time needed to finish the full handshake in the previous version. Considering a typical HTTPS use-case, two round trips were needed (see Figure 1.3) which can delay the connection noticeably, especially in higher latency environments, for example in mobile networks. And this problem has been neglected for several years [17].

TLS 1.3, however, has made a big change. The formerly used two round trip handshake evolved into just one round trip (see Figure 1.3) [17].

At the start of a communication, a `ClientHello` is sent to a server. The big difference is that the message contains a "Key share" (a session ID or a session ticket). In the "Key share" the client tries to guess what key agreement protocol could be used. Unless the guess is wrong, the server responds with a normal reply and in addition, the server is able to send handshake messages, including the `Finished` message inside the first response. This helps with solving the timely handshake in high-latency environments. If the client's guess is not correct or provides insufficient information, a `HelloRetryRequest` message is sent by the server and the full TLS handshake is executed [17].

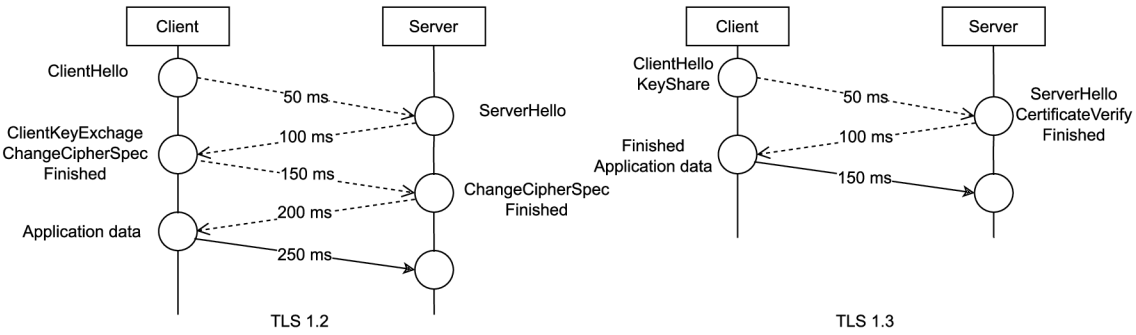

Fig. 1.3: Full TLS 1.2 and 1.3 handshake – comparison

Zero Round Trip Time resumption (0-RTT)

It is not possible to renegotiate a session in the new version like it used to be in the previous one. Instead, it is possible to authenticate a known server–host connection with 0-RTT. This is possible with the knowledge gathered from a previous session. When a client is connected to a server for the first time, using TLS 1.3, they agree on a resumption key. This is a so-called pre-shared key (PSK). This key is used

to encrypt the first block of application data (see Figure 1.4) during the first sent packet of the 0-RTT resumption. This, however, comes with the cost, that the first message from the client is prone to replay attacks and the key is from the previous session, therefore forward secrecy is not present in this step [8].

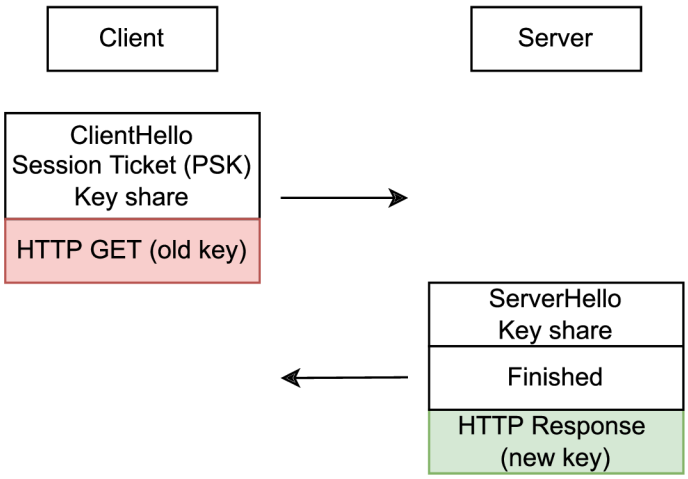

Fig. 1.4: 0-RTT resumption handshake

2 Software development

In the early years of software, software development quickly became a problematic area. The code developed by ad-hoc development[1] became quickly hard to maintain and bug-fix. From what is known even today, software should be developed incrementally and checked for bugs[2] as soon as possible [18].

2.1 Software Development Life Cycle

Software development is a complex process, therefore, it is crucial that the process is organized properly. Software Development Life Cycle (SDLC) helps to make this process controlled and clear. The most popular models of SDLC are:

- Waterfall Model,
- Iterative Model,
- Spiral Model,
- V-Model,
- Big Bang Model.

All the models mentioned above consist of similar stages:

1. **Planning and requirement analysis** – input from customers is analyzed by team members, sales and experts of that field; a basic plan is created and risks are minimized.
2. **Defining requirements** – Software Requirement Specification (SRS) is created; it is summarised what will be developed.
3. **Design product architecture** – from the SRS is a Design Document Specification (DDS) created and reviewed.
4. **Developing the product** – actual code is built.
5. **Testing the product** – the software is tested in-depth, if any bugs are present, the code goes back to the previous stage.
6. **Deployment** – if the product is flawless, it gets deployed.
7. **Maintenance** – the product is held up to security standards, bugs are being fixed, and possible upgrades can be done [19].

This is the usual life cycle of software from its start to its deployment and kept maintenance.

[1] No formal guidelines or process.
[2] Problems in software.

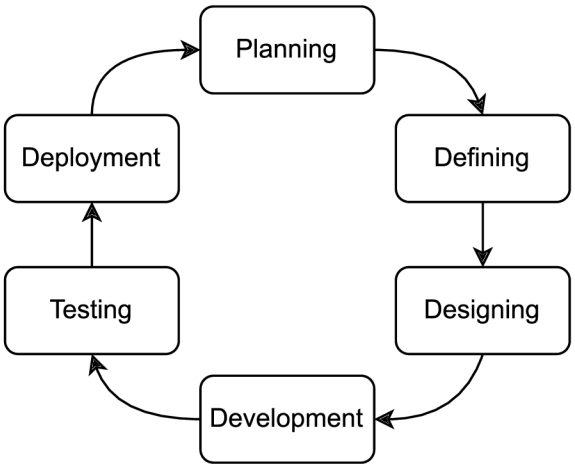

Fig. 2.1: Software Development Life Cycle

2.2 Software Testing Life Cycle

Software testing is such a large and important stage that a dedicated life cycle exists. In the testing stage, it is important to know what is the right amount of tests needed. As it is improper to under-test or to over-test software. The right amount is shown in Figure 2.2 [20].

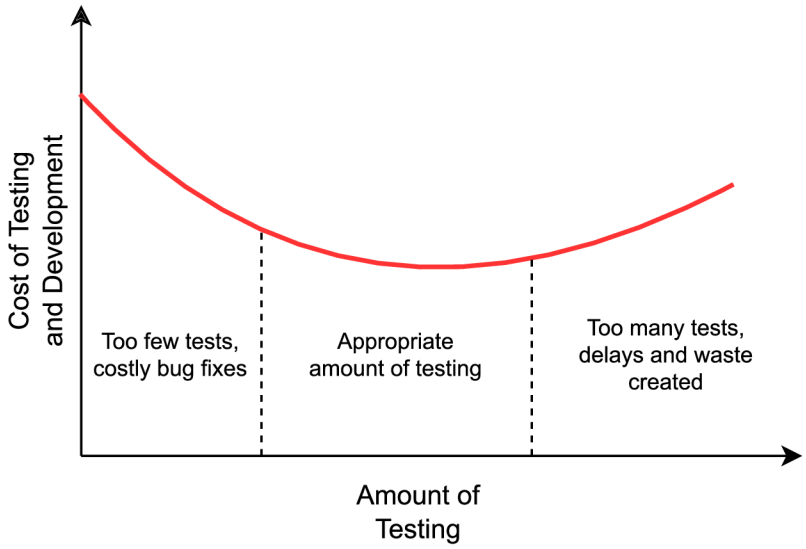

Fig. 2.2: The dependence of the amount of testing on the price

The initial stages of this life cycle strongly resemble the stages of SDLC in Section 2.1, however, the subsequent stages are unique to this life cycle.

1. **Requirement analysis** – all requirements are defined for the tests.
2. **Test planning** – the second most important stage; all test strategies are defined and a test plan document is created.

3. **Test case development** – test cases are created and expected results are defined. If needed, test data are created.
4. **Environment setup** – smoke tests can be executed in order to check the environment is set up correctly.
5. **Test execution** – the execution is based on the test plan. The tests pass or fail and are handled accordingly; partial reporting occurs.
6. **Test closure** – if all tests passed, reports, matrix and results are documented [21].

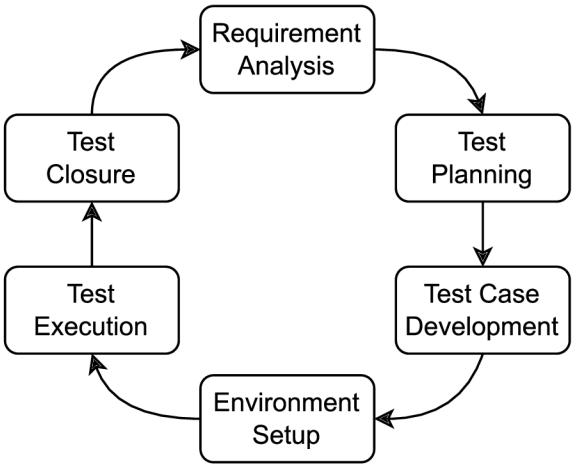

Fig. 2.3: Software Testing Life Cycle

2.3 Continuous Integration

Continuous Integration is a development practice that enhances the process of software development. This approach brings automation to the initial parts of the development process. It enables especially larger groups of developers to deliver a working code of higher quality and in a shorter time. The main point of CI is to build and check if the new code works right when it is checked in. With this in mind, developers and testers are able to deliver code with fewer bugs and more effectively. It can be very time-consuming to review a code that was written weeks or months ago. The necessary process of getting familiar with the code is an unnecessary waste of time and needs to be prevented. With CI, the feedback is faster and corrections can be applied immediately. Another beneficial effect that can CI bring is to check the quality of the code and if best practices of code writing are followed. This is especially important in projects, where multiple developers work together. All in all, with this methodology, developers are encouraged to commit daily so the changes are integrated more quickly [22, 23].

2.4 Countinuous Delivery and Deployment

Continuous Delivery is another development practice that is related to the previously mentioned process. As long as the CI process is successful, the need for the actual release of software is present as well. Release of software should be only done at times when the software was thoroughly tested and everything meets the expectations. However, an interesting phenomenon was noticed. Organizations that deliver (release) more frequently, create more reliable software. This is possible due to the CD methodology. Despite the fact that the release of software can be a problematic process, in CD this process is made easier and repeated many times throughout the development. This accomplishes that the process of delivering the software to the customer is already reliable. In addition to that, if anything fails during the process, a rollback can be easily executed.

However, this process still needs human intervention to trigger a deployment. This can be solved with Continuous Deployment. It makes the delivery automated and no additional intervention is needed. The key outcome of these automation processes CI/CD is to decrease space for human error and deliver better software [24, 22, 25].

3 Description of important tools/libraries

The process of integrating tests is complex and requires the use of various tools and libraries. This chapter describes the most important particularities that are to be encountered in the practical part.

3.1 GitLab CI/CD

In GitLab, the CI/CD methodologies are represented by pipelines. These pipelines are configured in a file ".gitlab-ci.yml" which resides in the root of a GitLab repository with the rest of the files. This file contains all the scripts which are executed with the dependencies important for the execution. The particular scripts are grouped into jobs, which in larger pipelines can be structured in different stages (e.g., Bootstrap, Build, Test). With a structured pipeline, jobs can be run in parallel independently. A start of a pipeline can be triggered for various reasons. Usually, it is triggered by a commit or a created merge request in a repository [26].

Jobs are executed by GitLab Runners. GitLab Runner is an application written in Go[1] which is connected to a GitLab repository. The Runner can be either provided by GitLab or can be installed on private infrastructure. It can be installed on several operating systems (GNU/Linux, macOS, and Windows) and jobs can be run locally, using e.g., Docker[2] or connecting by SSH (Secure Shell) [26].

3.2 Zuul

Zuul is another project gating system. This term stands for a CI and CD system that focuses on preventing breakage of the mainline[3]. It tests the submitted code to the repository against the main branch. The changes can be merged only if all tests pass.

Zuul is a complex system that provides resources to run jobs. In Zuul this is done by a separate component called Nodepool[4] [27].

3.3 Tools

This section describes the main tools which are present in the process of the test execution process. The main ones are TMT (Test Management Tool) and the main

[1] Programming language introduced by Google
[2] Platform for running containerized applications.
[3] main (production) branch of a repository
[4] Nodepool documentation

library which extends the test scripts – BeakerLib. These are Red Hat's open-source projects. Some tests may be dependent on other additional libraries for specific test cases.

3.3.1 Test Management Tool

TMT is a complex tool that is used to manage and execute tests. This tool can create tests, run them, review, and debug them. It is designed to be usable in CI systems.

Metadata

TMT uses metadata in `fmf` (Flexible Metadata Format) format. The format is derived from `yaml` format which means, it is human and machine-readable. These metadata define how the tests are handled when executed [28].

- **Level 0: Core** – this type of metadata can define `summary`, `description`, and few other less commonly used attributes. Attributes from this level can be used across all levels [29].
- **Level 1: Tests** – this metadata is stored closely to the test code. Mostly used attributes of this layer are the `test` script, `component`, `contact` and `require` [30].
- **Level 2: Plans** – this level of metadata is usually used, to define groups of desirable tests. Frequent attributes of this layer are `discover`, `prepare`, `execute`. Plans can even be imported from remote repositories [31].
- **Level 3: Stories** – this layer enables developers to track implementation, test and documentation coverage [32].

Pre-test phase

Before the tests are executed, the preparation has several phases. At first, the tests are discovered (see Listing 3.1).

Listing 3.1: Discover tests

```
$ tmt run discover --how=fmf
```

The host is provisioned (see Listing 3.2).

Listing 3.2: Provision machine

```
$ tmt run provision --how=local
```

Additional configuration is done to the test environment (see Listing 3.3).

Listing 3.3: Additional adjustments to environment

```
$ tmt run prepare
```

Test phase

Test execution is divided into several steps `discover`, `provision`, `prepare`, `execute`, `report` and `finish`. Running all steps can be triggered with a flag `all` (see Listing 3.4) [33].

Listing 3.4: Execute tests

```
$ tmt run --all
```

When the tests are executed, the test environment can be set. The environment can be provisioned in a virtual machine or creating a virtual machine using test cloud, on a local host, in a container, using a virtualization platform OpenStack or connect to a already provisioned machine using ssh [34]. The tests can also be run in a so-called interactive mode. This enables a tester to step into the environment and interact directly with the running instance of the test (see Listing 3.5).

Listing 3.5: Interactive mode

```
$ tmt run --all execute --how tmt --interactive
```

Another way of debugging tests which TMT provides is a hands-free debugging mode. This mode enables the test to be rerun without a tester's action. It enables the tester to edit the source file and watch the updated execution results automatically (see Listing 3.6).

Listing 3.6: Debug mode

```
$ tmt run debug
```

If additional information are needed, the verbosity (see Listing 3.7) and debug level can be raised (see Listing 3.8).

Listing 3.7: Verbosity increased

```
$ tmt run -v
```

Listing 3.8: Debug

```
$ tmt run --debug
```

3.3.2 BeakerLib

BeakerLib is a simple bash library. This library provides extended functionalities to the tests. It mainly simplifies creating, running, or analysis of tests. The main

advantage of using this library in tests is that it provides a good structure of the logs [35]. The logs can be divided into several phases:

- title of a test,
- description of a test,
- setup of a test,
- several phases of a test itself,
- cleanup after a test is completed,
- results.

An example of a Beakerlib phase is shown in Listing 3.9 and leads to a log shown in Listing 3.10.

Listing 3.9: Beakerlib – setup phase

```
...
rlPhaseStartSetup
  rlAssertRpm $PACKAGE
  rlRun "TmpDir=\$(mktemp -d)" 0 "Creating tmp directory"
  rlRun "pushd $TmpDir"

rlPhaseStartTest "Generate key and cert"
  rlRun "mkdir db"
...
```

Listing 3.10: Beakerlib logs of a setup phase

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Setup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 03:36:17 ] :: [   PASS   ] :: Checking for the presence
of gnutls rpm
:: [ 03:36:17 ] :: [   LOG    ] :: Package versions:
:: [ 03:36:17 ] :: [   LOG    ] ::  gnutls-3.7.2-2.fc35.x86_64
:: [ 03:36:17 ] :: [   PASS   ] :: Creating tmp directory
(Expected 0, got 0)
:: [ 03:36:17 ] :: [   PASS   ] :: Command 'pushd /tmp/tmp.
V7e9bWwP2M' (Expected 0, got 0)

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Generate key and cert
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 03:36:17 ] :: [   PASS   ] :: Command 'mkdir db'
(Expected 0, got 0)
```

3.3.3 Podman

Podman is an open-source tool. It is a container engine for development, management, and running of OCI[5] containers and container images on Linux systems. A container environment is also called an ecosystem. This ecosystem is solely managed by the libpod[6] library [36].

3.4 Security libraries

OpenSSL, GnuTLS and NSS (Network Security Services) are open-source cryptographic libraries written mainly in C[7]. The core of these projects deals with TLS and PKI stack. They implement cryptographic algorithms and make the implementation of security protocols easier. They are developed by different groups and distributed under various licences.

[5]Open Container Initiative
[6]libpod GitHub
[7]A programming language

3.4.1 OpenSSL

OpenSSL is maintained by a team of committers[8], and the whole project is run by the OpenSSL Management Committee[9]. It is licensed under the Apache License 2.0. The main advantage of OpenSSL is its complexity and performance, and it is also user-friendly to new developers. It mainly consists of three parts [37]:

- **libssl** – library with all TLS protocols up to version 1.3,
- **libcrypto** – a standalone library for the use of cryptographic algorithms in applications,
- **openssl** – this is a command-line tool that can be used to calculate hashes, create certificates, make client-server tests and many more.

3.4.2 GnutTLS

Besides the whole community[10], GnuTLS is mainly maintained by Tim Rühsen, Daiki Ueno and Dmitry Baryshkov. It mainly focuses on its implementation in Linux-based systems. An example of such an integration can be the GNOME desktop environment, which is very popular among Linux users. GnuTLS is licensed under LGPLv2.1+ (GNU Lesser General Public License) license and therefore, it is possible to use it in proprietary software [38].

3.4.3 NSS

NSS is a robust set of security libraries. It is designed in a way that enables the development of security-enabled client and server applications. It is maintained by Mozilla Foundation and is continually enhanced by individual contributors or organizations, and is available under Mozilla Public License. It is a copyleft license that permits combinations with other open or proprietary licenses. NSS is mainly integrated with products from Mozilla – Firefox, Thunderbird and other [39, 40].

[8] Members of OpenSSL committers
[9] Members OpenSSL management
[10] GnuTLS contributors

4 Test integration

In this chapter, the main focus is on practical part of the thesis. Because these tests are all related to Red Hat Enterprise Linux, which is a downstream[1] product, the interoperability tests should be executed first in the upstream[2] project. For this matter, the main aim are CentOS and Fedora distributions. In this thesis, the test integration is done in 3 projects – upstream GnuTLS, CentOS Stream and Fedora. There is one more project that is in progress (upstream OpenSSL), however, this project needed thorough expertise, and thus it has not been successfully finalized.

4.1 Local setup

Right at the start of the practical part, before anything is tested, it is required to create a suitable test environment for the tests. In this regard, there are multiple steps needed to be taken, for the best tester's experience. In this process of running tests locally, several modifications of the tests can be done, thanks to that TMT is a very modular and flexible tool. For simplicity, the initial setup was done on a local machine. To get the tests executable on the tester's machine there are several important actions to be taken. As mentioned before, the execution of interoperability tests requires a certain distribution of Linux. It would not be very effective to install a new bare metal machine only to run a few tests, therefore in the process of realization, containerization takes place, and using Podman (see Section 3.3.3) was found to be the best choice. When the Fedora container is up and running, the test tool which manages the tests needs to be installed there. Therefore, TMT (see Section 3.3.1) needs to be installed inside the container. The last essential part of the setup is to clone the needed tests from repositories and their dependencies to the container. This can be done in multiple ways. The approach which was used was mounting a folder from the container to the host machine.

This is the setup which leads a tester to debug the tests pretty comfortably and in a reasonable time on his own machine.

4.2 Test selection

The main goal of this bachelor's thesis is to take tests, which are executed in RHEL (various versions) downstream repositories and add them to the selected upstream projects.

[1] A released product by Red Hat that is paid.

[2] Development stream – opensource and free to use.

This is an approach which might be a little confusing, but it is actually beneficial for the development of TLS/SSL interoperability in the future when the tests are implemented into downstream products. The main benefit of this action is that the tests added to the upstream project can find problems sooner in the development process, which leads to a lower cost of development overall.

4.2.1 Local execution

When the initial setup is completed, it is possible to proceed with the local execution. First, it is needed to "exec"[3] into the test environment (see Listing 4.1).

Listing 4.1: Connect to a container interactively

```
$ podman exec -it fedora-35 /bin/bash
```

This command (see Listing 4.1) gives an interactive bash environment, thanks to the arguments `i` and `t`. If this command succeeds, the mounted folder containing the tests can be located. It is important to change the directory respectively (see Listing 4.2).

Listing 4.2: Change working directory

```
$ cd </path/to/test/directory>
```

When the working directory is adjusted the next command is run in order to execute them (see Listing 4.3).

Listing 4.3: TMT adjusted test executution

```
$ tmt run -a \
  -e "SLICE_TOTAL=$SLICES" \
  -e "SLICE_ID=$SLICEID" \
  plans -n interop tests -f "tag:interop-gnutls" \
  -f "tag:interop-$TYPE" -f "tag:interop-$COMPONENT" \
  provision -h local execute -h tmt --interactive
```

TMT runs with several arguments. All of them are present for a certain purpose.

- `run` – runs discovered test steps,
- `-a` – runs all steps (see Section 3.3.1); with this option it is possible to customize stages further,
- `-e "SLICE_TOTAL=$SLICES"` and `"SLICE_ID=$SLICEID"` – parameters which enable tests to run in parallel. This needs to be supported in the tests themselves first.
- `plans -n` – selects plans with specific names, this case – interop.

[3]To enter the container.

- `-f` – applies advanced filters, enabling us to run specific tests. For this parameter, we can set a "key:value" pair defined in the ".fmf" files.
- `provision -h` – this command sets what test environment will be provisioned, in this case, the local host is used for execution.
- `execute -h` – run tests using a specific executor, in this case, the tests are executed in an internal TMT executor. The tests are run one by one on the guest.
- `--interactive` – interactive mode, does not capture output.

Debugging

The approach in the section above is the best setup for quick debugging or development. This procedure is mostly used during the initial part of the tests review, to check the functionality of tests separately.

If unexpected behaviour is detected and/or any fails are encountered, it is possible to debug the tests right when they are being executed. This can be achieved by interrupting the test with a bash shell (see Listing 4.4). Now it is possible to inspect the environment in real time and examine the particular problem encountered.

Listing 4.4: Debugging a test phase using bash

```
...
rlPhaseStartTest "nss server"
  /bin/bash # Run bash
  rlRun "$SERV -d sql:nssdb -n localhost4
  -V ssl3:tls1.2 -p 4433 >server.log
  2>server.err &" nss_pid=$!
  rlRun "rlWaitForSocket -p $nss_pid 4433"
...
```

4.3 GnuTLS upstream

The first project this thesis focused on was the upstream GnuTLS library. Upstream project source code is located in a GitLab repository and the native CI system (GitLab CI/CD) is there configured.

When the tests are successful locally, they are ready for the next test stage. They are pushed to a personal fork of the upstream repository and the pipeline is executed. Optionally, new branches can be created and the ".gitlab-ci.yml" file is adjusted, so that only particular jobs are run. This speeds up the whole pipeline duration and results can be seen more quickly. In this stage, the tests are executed in the native GitLab CI system (see Section 3.1). Usually, the results of tests are

identical to those executed locally. However, as it was possible to experience during the testing, the result can divert from expectations. If that happens, naturally, the logs are inspected first in the search for possible indications of why the tests failed. If the problem is not easily discovered, tests can be put aside for further analysis.

4.3.1 Local GitLab Runner

During the course of this project, the tests were executed on GitLab's infrastructure. However, this can become limiting, because runners can have time limitations. This is why there was an interest in hosting a runner locally. Since the pipeline is configured in a way that a Linux image is pulled, the runner needed to be hosted using the Docker executor.

Runner setup

A runner needs to be installed first, subsequently, it is registered to a particular project. The runner container can be deployed with a command shown in Listing 4.5.

Listing 4.5: GitLab Runner intallation

```
$ docker run -d --name gitlab-runner --restart always \
-v /srv/gitlab-runner/config:/etc/gitlab-runner \
-v /var/run/docker.sock:/var/run/docker.sock \
gitlab/gitlab-runner:latest
```

Afterwards, the runner needs to be registered to a particular project (see Listing 4.6).

Listing 4.6: GitLab Runner registration

```
$ docker run --rm -it -v /srv/gitlab-runner/config:
/etc/gitlab-runner gitlab/gitlab-runner register
```

With this command, the runner asks for additional information:

1. GitLab instance URL,
2. registration token,
3. description,
4. tags – they are needed to start jobs with tags,
5. maintenance information,
6. runner executor – for this instance, Docker is used,
7. default image – if a job does not specify an image, not relevant for these tests.

If there is a need to additionally change the runner's configuration, a configuration file "config.toml" can be adjusted.

Problems encountered

This process was successful to a certain extent. The runner was visible from the GitLab project perspective, and it was even possible to run pipelines in it. The first several runs looked promising and the tests passed as expected. Unfortunately, during the course of this bachelor's thesis, the runner became inconsistent in its test results. Later, it was discovered, those anomalies appeared due to persistent cache from previous runs.

Another problem encountered later, during testing, was the inability to pull submodules[4] of the tested repository. This root cause was discovered very late in the process and was not solved.

Finally, successful tests showed, the locally hosted GitLab runner can create a heavy workload on the local machine. That is why a move to a cloud solution was eventually preferred. There would be further studies needed, on how to configure pipelines with present submodules on non-GitLab infrastructure properly.

4.4 CentOS Stream

When working on the CentOS Stream, reliable tests were already found. Therefore, the intensive test selection part is skipped from now on.

This project was interesting because it actually meant a contribution to packages of GnuTLS, OpenSSL, and NSS in CentOS. Even though the project resides on GitLab, the underlying CI infrastructure is actually maintained by Red Hat. For that reason, the widely used open-source tool Zuul CI is responsible for the pipelines and test execution. The test execution is further manageable by a bot with additional functionalities.

4.4.1 Repository setup

To make the repository accessible by TMT from Zuul's perspective, a folder ".fmf" and a file "version" had to be added to the root of the repository. The file usually contains only one character: "1". When these additions are made, the repository becomes visible to the CI and tests can be run. But now, the approach is a bit different than it was before. At this point, it is neither necessary nor wanted, to use a "git submodule" of the repository, where tests physically reside. That is because the test execution is defined differently. The best way how to add tests to this repository is to use L2-level metadata – plans.

[4] Additional repositories in a git repository.

Multiple plans are created and they define which tests are executed. Individual plans are run in parallel and tests in one plan are executed sequentially.

Example of an added plan in Listing 4.7. This plan filters out only one test that is needed to run in parallel, for its long runtime.

Listing 4.7: Plan for gnutls_tls-1-3-interoperability-gnutls-nss-2way

```
summary: Upstreamed gnutls-openssl interop-2way tests
contact: Stanislav Zidek <szidek@redhat.com>
discover:
  # upstreamed tests (public)
  - name: interop-nss-2way
    how: fmf
    url: https://gitlab.com/redhat-crypto/tests/interop.git
    filter: 'tag: interop-gnutls & tag: interop-nss &
             tag: interop-parallel-1'
execute:
    how: tmt
```

After creating plans similar to above, all the desired tests are eventually added to the repositories. Then, the process of merging a change to the main branch can follow. Specifically, contribution to CentOS follows a basic PR workflow[5] and the commit message needs to declare an author (see Listing 4.8).

Listing 4.8: Example of declaring an author

```
$ git commit -m "Fix contact information" --sign-off
```

Additionally, an official bug has to be raised by the developer. This is a way to make an official request for a change in a product. Subsequently, the bug is checked and gets approval (or denial) from a maintainer. The final commit message should look similar to that shown in Listing 4.9. After these steps are completed, a pipeline that checks the correctness of the requirements mentioned should finish successfully. Now the branch is ready to be merged.

Listing 4.9: Explicit declaration of author, and internal bug number with a commit message.

```
Fix contact information
Related: rhbz#2180023
Signed-off-by: Peter Leitmann <pleitman@redhat.com>
```

This process is relevant for the three CentOS projects – GnuTLS, OpenSSL and NSS. All the plans, which were added, can be seen in the electronic appendix.

[5]Pull Request workflow means, that a fork of a repository is made and then a pull/merge requested is created towards the original repository.

4.5 Fedora

A very similar approach to the one in CentOS is also used in Fedora project (except for the required official bugs and explicit declaration of the author – these are not necessary). The repository is again served by Zuul CI and ".fmf" folder has to be added similarly.

Plans for the particular components are already created, so under ideal circumstances, the tests could been added straightaway. However, after some time of the implementation process, `--filter` parameters became unbearably complex. That is why many tests underwent a process of adding unique tags to simplify the filtering.

Fedora repository

The tricky part about contributing to the Fedora project is that the project is hosted in Pagure[6], so it also requires the use of a tool "fedpkg"[7] to push the changes to the remote repository (see Listing 4.10).

Listing 4.10: fedpkg push to the remote repository

```
$ fedpkg push
```

4.6 OpenSSL upstream

The fourth project to contribute to is the upstream OpenSSL library. It appeared to be a challenge right from the start. The project's code is hosted on GitHub and heavily relies on GitHub Actions[8].

The first proposal of implementation of the tests was using a GitHub Actions service called "Packit". This service is developed by Red Hat and can be used to run tests relatively easily. On the other hand, to make the build of the repository work, the addition of a spec file[9] would be necessary (that represented a huge complication), and after the initial discussion with maintainers, this approach was ruled out.

Maintainers prefer using GitHub Actions, thus a new workflow file had to be created. Initially, this seemed to be a complicated task, because of little to no experience with writing "workflow" files[10]. However, things eventually got complicated for a different reason. The native GitHub runners support only a few operating

[6] Fedora Pagure documentation
[7] fedpkg
[8] Native CI system in GitHub.
[9] A file that defines steps, how a package is built.
[10] Files that define jobs in GitHub Actions

systems, while none of them was any RPM (Red Hat Package Manager) based distribution (Ubuntu). This fact opened a new discussion.

Unfortunately, running selected TMT tests on Ubuntu is not yet possible due to the absence of the tool's ability to download test-required dependencies.

The best way how to handle this obstacle turned out to be a creation of a Fedora container on top of Ubuntu. This was not as problematic as dealing with the GitHub Actions' TTY[11] after all. Because a GitHub runner does not use a TTY, commands that would be used in other environments did not generate any command line output (see Listing 4.11 (logs)).

Listing 4.11: Example of running Podman with a warning message

```
$ podman run -it <image> <command>

msg="The input device is not a TTY. The --tty and
--interactive flags might not work properly"
```

Since logs are a critical aspect of a pipeline, this had to be fixed. Therefore, the container had to be first run without `--tty` (`-t`) flag, and the rest of the commands had to be executed afterwards also without the same flag (see Listing 4.12).

Listing 4.12: Not interactive TTY – workaround

```
$ podman run -d --name fedoralatest fedoralatest:latest
$ echo "Tests to run:"
$ podman exec -i fedoralatest tmt run plans -n$COMPONENT
discover -v
$ echo "Run the tests:"
$ podman exec -i fedoralatest tmt run -a plans -n$COMPONENT
provision -h local execute -h tmt --interactive
```

[11] Teletypewriter – input/output device

Conclusion

This bachelor's thesis was focused on the comprehension of the theoretical knowledge of Transport Layer Protocol and code testing methodologies.

In the first chapter, the history of SSL and TLS protocol is described followed by a thorough description of how the TLS protocol operates and how it is constructed. Moreover, a comparison of the last two releases of the TLS protocols was conducted.

Furthermore, the second chapter explains, what processes lay behind software development. Firstly a description of the Software Development Life Cycle was made. Which mainly consists of planning, designing the product, development, deployment and maintenance.

In addition to that, software needs to undergo testing, too. This is why the Software Testing Life Cycle is described as another important part of the development process, which mainly consists of considering what things are to be tested and the way how these tests should be carried out.

In the later section are explained the advantages of implementing CI/CD processes into the Software Development Life Cycle. These technologies mainly create consistency in testing and deployment. The third chapter is the last theoretical part. It covers the theory about the main tools and libraries that are necessary for test execution and implementation.

The practical part of the thesis starts with the description of the steps for the test selection and follows with the test integration into the first project – GnuTLS upstream. In this project, some of the tests were already present, but many of them were added during this thesis too. Moreover, some minor changes were done to the CI configuration file which meant better performance of some tests.

Afterwards, tests were added to the other three selected projects. Starting with the CentOS Stream and Fedora, where Zuul CI is the underlying infrastructure. In these two operating systems, several tests were added to all desired projects: GnuTLS, OpenSSL and NSS. The first two projects needed to be made visible to TMT in Zuul by adding the ".fmf" directory, and plans had to be created that would handle a correct test selection and execution.

The last part of the thesis covers the process of contributing to an open-source project – OpenSSL. This was one of the most challenging parts for a number of reasons described in that section. First and foremost, the most interesting was the needed discussion with the maintainers of such a big project, but also the initial absence of experience with GitHub Actions.

Finally, all the objectives of this thesis were successfully accomplished. In addition, a fourth project was brought into a development state and is expected to be completed in the near future.

References

1. RISTIĆ, I. *Bulletproof SSL and TLS*. 1st ed. London: Feisty Duck Limited, 2015. ISBN 978-1-907117-04-6.
2. TURNER, S. *Prohibiting Secure Sockets Layer (SSL) Version 2.0* [online]. RFC Editor, 2011 [visited on 2022-11-09]. RFC, 6176. RFC Editor. ISSN 2070-1721. Available from: `https://www.rfc-editor.org/rfc/rfc6176`.
3. BARNES, R.; THOMSON, M.; PIRONTI, A.; LANGLEY, A. *Deprecating Secure Sockets Layer Version 3.0* [online]. RFC Editor, 2015 [visited on 2022-10-01]. RFC, 7568. RFC Editor. ISSN 2070-1721. Available from: `http://www.rfc-editor.org/rfc/rfc7568.txt`.
4. *Cryptographic security protocols: TLS* [online]. IBM, 2022. [visited on 2022-10-19]. Available from: `https://www.ibm.com/docs/en/ibm-mq/9.1?topic=mechanisms-cryptographic-security-protocols-tls`.
5. DIERKS, T. *Security Standards and Name Changes in the Browser Wars* [online]. [visited on 2022-10-16]. Available from: `https://tim.dierks.org/2014/05/security-standards-and-name-changes-in.html`.
6. NOHE, P. *TLS 1.3: Everything you need to know* [online]. Hashed Out, 2019. [visited on 2022-10-11]. Available from: `https://www.thesslstore.com/blog/tls-1-3-everything-possibly-needed-know/`.
7. DIERKS, T.; RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.2* [online]. RFC Editor, 2008 [visited on 2022-10-02]. RFC, 5246. RFC Editor. ISSN 2070-1721. Available from: `http://www.rfc-editor.org/rfc/rfc5246.txt`. `http://www.rfc-editor.org/rfc/rfc5246.txt`.
8. RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.3* [online]. RFC Editor, 2018 [visited on 2022-10-07]. RFC, 8446. RFC Editor. ISSN 2070-1721.
9. *CVE-2012-4929* [online]. CVE Details, 2012. [visited on 2022-10-07]. Available from: `https://www.cvedetails.com/cve/CVE-2012-4929/`.
10. OPPLIGER, R. *SSL and TLS Theory and Practise*. 2nd ed. Massachusetts: Artech House, 2016. ISBN 978-1-60807-998-8.
11. *An overview of the SSL/TLS handshake* [online]. IBM, 2022. [visited on 2022-10-16]. Available from: `https://www.ibm.com/docs/en/ibm-mq/9.1?topic=tls-overview-ssltls-handshake`.

12. MUNSHI, A. *TLS v1.2 handshake overview* [online]. Medium. [visited on 2022-10-16]. Available from: https://medium.com/@ethicalevil/tls-handshake-protocol-overview-a39e8eee2cf5.

13. WARBURTON, D. *The 2021 TLS Telemetry Report* [online]. F5 Labs. [visited on 2022-10-28]. Available from: https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report.

14. GERACI, A. *TLS v1.3 Mandates Perfect Forward Secrecy – Make Sure You're Prepared!* [Online]. WorldTechIT, 2018. [visited on 2022-10-10]. Available from: https://wtit.com/tls-v1-3-mandates-perfect-forward-secrecy-pfs-make-sure-your-f5-is-prepared/.

15. SULLIVAN, N. *A Detailed Look at RFC 8446 (a.k.a. TLS 1.3)* [online]. Cloudflare, 2018. [visited on 2022-10-11]. Available from: https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/.

16. BRODIE, A. *Overview of TLS v1.3* [online]. OWASP. [visited on 2022-10-10]. Available from: https://owasp.org/www-chapter-london/assets/slides/OWASPLondon20180125_TLSv1.3_Andy_Brodie.pdf.

17. THAKKAR, J. *TLS 1.3 Handshake* [online]. Hashed Out, 2018. [visited on 2022-11-04]. Available from: https://www.thesslstore.com/blog/tls-1-3-handshake-tls-1-2/.

18. MILLS, H.D. Software Development. *IEEE Transactions on Software Engineering*. 1976, vol. SE-2, no. 4, pp. 265–273. Available from DOI: 10.1109/TSE.1976.233831.

19. VELIMIROVIC, A. *What is SDLC? Understand the Software Development Life Cycle* [online]. PhoenixNAP, 2022. [visited on 2022-10-15]. Available from: https://phoenixnap.com/blog/software-development-life-cycle.

20. PITTS, Ch. *Under-and Over-Testing: Problems* [online]. Hexawise, 2022. [visited on 2022-10-16]. Available from: https://hexawise.com/posts/oasdkhiad.

21. HAMILTON, T. *STLC Phases, Entry, Exit Criteria* [online]. Guru99, 2022. [visited on 2022-10-16]. Available from: https://www.guru99.com/software-testing-life-cycle.html.

22. ROSSEL, S. *Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment*. Packt Publishing, 2017. ISBN 9781787284180.

23. MEYER, Mathias. Continuous Integration and Its Tools. *IEEE Software*. 2014, vol. 31, no. 3, pp. 14–16. Available from DOI: 10.1109/MS.2014.58.

24. HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st ed. Massachusetts: Addison-Wesley Proffesional, 2010. ISBN 0321601912.

25. CHEN, Lianping. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*. 2015, vol. 32, no. 2, pp. 50–54. Available from DOI: 10.1109/MS.2015.27.

26. *Gitlab CI/CD* [online]. GitLab, 2022. [visited on 2022-10-26]. Available from: https://docs.gitlab.com/ee/ci/.

27. *Zuul* [online]. Read the Docs. [visited on 2022-10-30]. Available from: https://zuul-ci.org/docs/zuul/latest/index.html.

28. *Test Management Tool* [online]. Red Hat. [visited on 2022-11-11]. Available from: https://tmt.readthedocs.io/en/stable/index.html.

29. *TMT – Level 0: Core* [online]. Red Hat. [visited on 2023-04-10]. Available from: https://tmt.readthedocs.io/en/stable/spec/core.html#spec-core.

30. *TMT – Level 1: Tests* [online]. Red Hat. [visited on 2023-04-10]. Available from: https://tmt.readthedocs.io/en/stable/spec/tests.html.

31. *TMT – Level 2: Plans* [online]. Red Hat, [n.d.]. [visited on 2023-04-10]. Available from: https://tmt.readthedocs.io/en/stable/spec/plans.html.

32. *TMT – Level 3: Stories* [online]. Red Hat. [visited on 2023-04-10]. Available from: https://tmt.readthedocs.io/en/stable/spec/stories.html.

33. *TMT – steps* [online]. Red Hat. [visited on 2023-05-10]. Available from: https://tmt.readthedocs.io/en/stable/examples.html#select-steps.

34. *TMT – provision-machine* [online]. Red Hat. [visited on 2023-05-10]. Available from: https://tmt.readthedocs.io/en/stable/examples.html#provision-options.

35. *BeakerLib* [online]. GitHub. [visited on 2022-10-24]. Available from: https://github.com/beakerlib/beakerlib.

36. *Podman* [online]. Podman. [visited on 2022-10-24]. Available from: https://docs.podman.io/en/latest/.

37. *OpenSSL* [online]. GitHub. [visited on 2023-05-10]. Available from: https://github.com/openssl/openssl.

38. *GnuTLS main page* [online]. GnuTLS. [visited on 2023-05-12]. Available from: https://www.gnutls.org/.

39. *NSS* [online]. Unified XUL Platform MDN Backup. [visited on 2023-05-10]. Available from: `https://udn.realityripple.com/docs/Mozilla/Projects/NSS`.

40. *NSS – Ubuntu* [online]. Canonical Ltd. [visited on 2023-04-12]. Available from: `https://ubuntu.com/server/docs/network-security-services-nss`.

List of symbols and abbreviations

0-RTT Zero Round Trip Time.
AEAD Authenticated Encryption with Associated Data.
anon Anonymous.
CD Continuous Deployment/Delivery.
CI Continuous Integration.
CI/CD Continuous Integration and Continuous Deployment/Delivery.
CPU Central Processing Unit.
DDS Design Document Specification.
DH Diffie–Helman.
DHE Diffie–Hellman Ephemeral.
DSS Digital Signature Standard.
ECDH Elliptic-curve Diffie–Hellman.
FMF Flexible Metadata Format.
HTTPS Hypertext Transfer Protocol Secure.
ID Identification.
IETF Internet Engineering Task Force.
ISO/OSI International organization of Standardization/Open System Interconnection.
L2 Level 2.
LGPL GNU Lesser General Public License.
MAC Message Authentication Code.
MD5 Message Digest Algorithm 5.
NSS Network Security Services.
OCI Open Container Initiative.
PKCS Public-Key Cryptography Standards.
PKI Public Key Infrastructure.
PR Pull Request.
PRF Pseudorandom function.
PSK Pre-shared key.
RHEL Red Hat Enterprise Linux.
RPM Red Hat Package Manager.
RSA Rivest Shamir Adleman algorithm.
RSA-PSS RSA-Probabilistic Signature Scheme.
SDLC Software Development Life Cycle.

SRS Software Requirement Specification.

SSL Secure Sockets Layer.

TCP/IP Transmission Control Protocol over Internet Protocol.

TLS Transport Layer Security.

TMT Test Management Tool.

TTY Teletypewriter.

A Summary of pull requests

A.1 GnuTLS upstream

Added new interoperability tests:
https://gitlab.com/gnutls/gnutls/-/merge_requests/1680
Add new interop tests:
https://gitlab.com/gnutls/gnutls/-/merge_requests/1702

A.2 CentOS Stream

Merge requests for CentOS Stream.

A.2.1 GnuTLS

Add upstream interop tests:
https://gitlab.com/redhat/centos-stream/rpms/gnutls/-/merge_requests/58
Fix contact information:
https://gitlab.com/redhat/centos-stream/rpms/gnutls/-/merge_requests/67
Update filtering, minor changes:
https://gitlab.com/redhat/centos-stream/rpms/gnutls/-/merge_requests/68

A.2.2 OpenSSL

Add interop rpm-tmt-tests:
https://gitlab.com/redhat/centos-stream/rpms/openssl/-/merge_requests/96

A.2.3 NSS

Add new interop rpm-tmt-tests:
https://gitlab.com/redhat/centos-stream/rpms/nss/-/merge_requests/43

A.3 Fedora

Pull requests for Fedora.

A.3.1 GnuTLS

Add TMT interop tests:
`https://src.fedoraproject.org/rpms/gnutls/pull-request/83`

A.3.2 OpenSSL

Add TMT interoperability tests:
`https://src.fedoraproject.org/rpms/openssl/pull-request/43`

A.3.3 NSS

Add TMT interoperability tests:
`https://src.fedoraproject.org/rpms/nss/pull-request/26`

B Electronic attachments

```
/ ..... root directory of archive
├── gnutls-upstream ..... patches applied in GnuTLS upstream
│   ├── gnutls-upstream#1.patch ..... merge request number 1
│   └── gnutls-upstream#2.patch ..... merge request number 2
├── openssl-upstream ..... file to be added to OpenSSL upstream
│   ├── openssl-upstream-initial-setup.yml ..... first proposal
│   └── openssl-upstream-latest-setup.yml ..... latest proposal to date 24.5.2023
└── plans-centos+fedora ..... added plans to CentOS and Fedora
    ├── gnutls ..... Files for GnuTLS in both projects
    │   ├── .fmf .....
    │   │   └── version .....
    │   ├── ci.fmf .....
    │   ├── nss-2way.fmf .....
    │   ├── openssl-2way.fmf .....
    │   └── short-interop-tests.fmf .....
    ├── nss ..... Files for NSS in both projects
    │   ├── .fmf .....
    │   │   └── version .....
    │   ├── ci.fmf .....
    │   ├── gnutls-2way.fmf .....
    │   ├── openssl-2way.fmf .....
    │   ├── openssl-reneg.fmf .....
    │   └── short-interop-tests.fmf .....
    └── openssl ..... Files for OpenSSL in both projects
        ├── .fmf .....
        │   └── version .....
        ├── ci.fmf .....
        ├── gnutls-2way.fmf .....
        ├── nss-2way.fmf .....
        ├── openssl-reneg.fmf .....
        └── short-interop-tests.fmf .....
```