



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SAIJA SORSA
PROTOCOL FUZZ TESTING AS A PART OF SECURE SOFTWARE
DEVELOPMENT LIFE CYCLE

Master of Science thesis

Examiners: Prof. Hannu-Matti Järvinen
Examiners and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
9th of August, 2017

ABSTRACT

SAIJA SORSA: Protocol Fuzz Testing as a part of Secure Software Development Life Cycle

Tampere University of Technology

Master of Science thesis, 59 pages

February 2018

Master's Degree Programme in Information Technology

Major: Pervasive Systems

Examiners: Prof. Hannu-Matti Järvinen

Keywords: Fuzz Testing, Fuzzing, Security, SDLC, Automated fuzz testing

During the last couple of years the importance of software security has gained a lot of press recognition and it has become very important part of different software products especially in the embedded industry. To prevent software security vulnerabilities the secure software development life cycle is recommended as a development method to prevent implementation bugs and design flaws in the early phase of the product development. Secure software development life cycle recommends various different security actions to be taken in different phases of the development life cycle. Fuzz testing is one of these recommendations to be taken under secure software development life cycle.

Fuzz testing is an automated testing technique where the system under test is given modified and malformed also known as fuzzed input data. The purpose of fuzz testing is to find implementation bugs and security related vulnerabilities. Fuzz testing has been proven to be cost effective method to identify such issues. To increase the effectiveness of fuzz testing, such methods can be directly included in the implementation phase of the secure software development life cycle. Since many software products are developed by using continuous integration methods and automated testing, the automated fuzz testing can be integrated to existing continuous integration environment to enable automated fuzz testing beside normal testing procedures.

The purpose of this thesis is to create a fuzz testing framework that can be integrated to the existing testing and continuous integration framework and enable automated fuzz testing in the earliest phase of secure software development life cycle as well as in the verification phase of secure software development framework. The purpose of fuzz testing is to find security related vulnerabilities in multiple different KONE proprietary protocols used in embedded environment and to ensure the proprietary

protocols are secure.

TIIVISTELMÄ

SAIJA SORSA: Protocol Fuzz Testing as a part of Secure Software Development Life Cycle

Tampereen teknillinen yliopisto

Diplomityö, 59 sivua

Helmikuu 2018

Tietotekniikan koulutusohjelma

Pääaine: Pervasive Systems

Tarkastajat: Prof. Hannu-Matti Järvinen

Avainsanat: Fuzz-testaus, Fuzzing, Tietoturvallisuus, SDLC, Automaattinen fuzz-testaus

Viime vuosien aikana tietoturvallisuus on saanut paljon mediajulkisuutta ja siitä on tullut tärkeä osa-alue erilaisille ohjelmistoille, etenkin sulautettujen järjestelmien ohjelmistoille. Tietoturvallinen ohjelmistokehitysmalli on suositeltu tietoturva-avoittuvuuksien- ja ongelmien estämiseksi. Tietoturvallinen ohjelmistokehitys pyrkii estämään toteutus- ja suunniteluvirheitä ohjelmistokehityksen mahdollisimman aikaisessa vaiheessa. Tietoturvallinen ohjelmistokehitys suosittelee erilaisten tietoturvatoimintojen tekemistä ohjelmistokehityksen eri vaiheissa. Fuzz-testaus on eräs näistä suositelluista toimenpiteistä.

Fuzz-testaus on automaattinen testausmenetelmä, missä testikohteelle syötetään muokattua ja virheellistä syötettä. Fuzz-testauksen tarkoituksena on löytää toteutusvirheitä ja tietoturvaluushaavoittuvuuksia. Fuzz-testaus on hyvin kustannustehokas testausmenetelmä löytämään näitä ongelmia. Jotta fuzz-testauksen tehokkuutta voidaan parantaa, sitä voidaan tehdä jo tietoturvallisen ohjelmistokehityksen varhaisessa kehitysvaiheessa. Koska monet ohjelmistot kehitetään jatkuvalla integraatiolla ja testataan käyttäen automaatiotestausta, automatisoitu fuzz-testaus voidaan lisätä jo olemassaolevaan jatkuvan integraation ympäristöön, jolloin fuzz-testausta voidaan suorittaa muun testauksen ohessa.

Tässä lopputyössä kehitettiin fuzz-testaus ympäristö, joka voidaan liittää jo olemassaolevaan testaus ja jatkuvan integraation ympäristöön. Tämä integraatio mahdollistaa automatisoidun fuzz-testauksen tietoturvallisen ohjelmistokehityksen verifiointi- ja implementointi vaiheissa. Fuzz-testauksen tarkoituksena on havaita KONEen kehittämien sulautettujen järjestelmien protokollien tietoturva-avoittuvuuksia.

PREFACE

This thesis was performed at KONE Corporations IoT R&D Hyvinkää security team during the end of the year 2017 as a subcontractor via Etteplan Design Center Oy.

I really enjoyed working at KONE corporations R&D software security team. I learnt a lot and had the best time ever at office. There is a long list of persons who I would like to personally thank to make this thesis happen. Here is the minimum viable listing of the persons I want to express my regards and who had a huge positive impact for this thesis, not in any specific order: Jussi Valkiainen, Onur Zengin, Erkki Laite, Sami Lehtinen, Joonas Kannisto, Markku Vajjaranta, Jippo aka "Jipsotin" and dear Aleksandr Tserepov-Savolainen. I would also want to express my gratefulness for Etteplan and KONE for giving me this great opportunity to finish my university studies. The most supportive persons for this project from Etteplan has been Laura Haimakainen and Mikko Lindström. I'm glad that I had the opportunity to work with both of you.

Since I have not had the opportunity to thank my previous Intel coworkers, I would like to acknowledge especially Miia Onkalo for career guidance and for being the best role model in my life. My gratitude goes to other Intel coworkers I also had.

The one who really made this thesis happen was Antonios Michalas, he was the one who gave it the finishing touch and I'm impressed by his ability to work fast and precise. He was able to be one of the best supervisors I have had by, first of all responding to my queries, making good suggestions to improve my thesis, and showing a genuine interest for my graduation.

Finally I would like to thank my mom for listening my endless phone calls regarding this writing process as well a couple of others bystanders. For the ones who were not explicitly mentioned here, know that you are in my thoughts. Graduating and receiving my 3rd university degree has been a dream of mine for years. I'm glad I have been able to achieve my dreams in such a young age.

Saija Sorsa

Hyvinkää, February 12, 2018

CONTENTS

1. Introduction	2
1.1 Requirements and restrictions	2
1.2 The scope of this thesis	3
2. Security Development Life Cycle	4
2.1 Security Training	9
2.2 Requirements	10
2.2.1 Business Impact Analysis	12
2.2.2 Risk Assessment	13
2.2.3 SQUARE	15
2.3 Design	18
2.3.1 Secure Architecture Review	19
2.3.2 Threat Modeling	20
2.3.3 Attack Surface Reduction	22
2.3.4 STRIDE	23
2.4 Implementation	24
2.4.1 Secure Code Review	25
2.4.2 Static Code Analysis	26
2.5 Verification	28
2.5.1 Penetration Testing	29
2.6 Release	30
3. Fuzz Testing Techniques	32
3.1 Black-box Fuzzing	35
3.1.1 PULSAR	36
3.2 Gray-box Fuzzing	36
3.3 White-box Fuzzing	37
3.3.1 American Fuzzy Lop	37
3.3.2 SAGE	38

4. Implementation of the Fuzz Testing Framework	40
4.1 Automated Software Testing tools	41
4.1.1 Robot Framework	41
4.2 Continuous Integration tools	42
4.2.1 Jenkins	43
4.3 Used Fuzz Testing tool	44
4.3.1 Fuzz Testing Tool Requirements	45
4.3.2 Radamsa Fuzzer	45
4.4 Fuzz Testing Framework Implementation	46
4.5 Fuzz Testing Findings	47
5. Possible Future Work	49
6. Conclusions	51

LIST OF FIGURES

2.1	Amount of software vulnerabilities during the years 1999-2017.	4
2.2	Quality of software consists of security, privacy and reliability.	5
2.3	Secure Development Life Cycle phases.	8
2.4	Security Training levels.	9
2.5	Risk Assessment process.	13
2.6	Threat Modeling and Secure Architecture Review in SDLC.	19
2.7	Attack Surface area by different access levels.	23
2.8	STRIDE threat taxonomy.	23
2.9	Complementary testing methods in SDLC.	28
2.10	Overview of Penetration Testing.	30
3.1	Fuzz Testing phases.	32
3.2	Program malfunctions found by fuzz testing.	33
3.3	Black-box Testing.	34
3.4	PULSAR Fuzzing method.	36
4.1	Fuzzing protocol stack.	41
4.2	Fuzz Testing input space.	43
4.3	Continuous Integration.	44

LIST OF TABLES

2.1	Original Microsoft SDLC phases.	7
2.2	Security training topics.	10
2.3	Security and Privacy Risk Assessment.	14
2.4	Privacy Impact Rating.	15
2.5	All the steps in SQUARE method.	18
2.6	DREAD threat rating method.	21
2.7	Example unsafe and safe C functions.	24
2.8	Example weaknesses found by static code analysis.	27
2.9	Steps to be taken to hack remote PC.	29
4.1	Fuzz testing findings.	48

LIST OF PROGRAMS

- 2.1 Example of a classic C buffer overflow with unsafe function **memcpy**.
The function **memcpy** does not verify that the data storing buffer is large enough to save the data and thus can lead to buffer overflow. . . 24
- 3.1 If x is 32-bit integer the else branch is executed randomly with probability of $\frac{1}{2^{32}}$ [77]. 38
- 4.1 Example usage of Radamsa. 46

LIST OF ABBREVIATIONS

AFL	American Fuzzy Lop
API	Application Programming Interface
BIA	Business Impact Analysis
BIC	Business Impact Category
CI	Continuous Integration
CIA	Confidentiality, Integrity and Availability
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DDoS	Distributed Denial-of-Service
DREAD	Risk Assessment Model
EU	European Union
Fuzzing	Fuzz testing, automated software testing technique
GDPR	General Data Protection Regulation
IETF	Internet Engineering Task Force
IoT	Internet of Things
MITM	Man-in-the-middle
NHS	National Healthcare System
NIST	National Institute of Standards and Technology
Nmap	The Network Mapper
SAFECODE	The Software Assurance Forum for Excellence in Code
SAGE	Scalable Automated Guided Execution
SDLC	Security Development Life Cycle
SSH	Secure Shell
SQL	Structured Query Language
STRIDE	Threat taxonomy
SUT	System Under Test
US-CERT	United States Computer Emergency Readiness Team
OUSPG	Oulu University Secure Programming Group
OWASP	Open Web Application Security Project
PRNG	Pseudo Random Number Generator

1. INTRODUCTION

KONE is one of the worlds leading elevator and escalator vendors and provides 24/7 connectivity to some of its products. These type of IoT (Internet of Things) connectivity services needs to be secure, since information security breaches have been increased during the last years. To prevent these, KONE develops its products to be secure.

KONE has various proprietary protocols used in its IoT products. The security and the robustness of the proprietary protocols needs to be verified as early as possible. Fuzz testing is an automated software security testing technique, which can be used to detect security issues. Including automated fuzz testing in the automated testing environment will help to find possible implementation issues and security vulnerabilities in the earliest phase of the development process.

The purpose of this work is to enable fuzz testing in secure software development life cycle implementation and verification phases in KONE R&D IoT environment. Incorporating fuzz testing prevents security related bugs and vulnerabilities as soon as possible in the early phase of the development. The fuzz testing needs to be automated and be part of regression testing for verifying that the found issues are fixed properly. This thesis, consists of the description of secure development life cycle phases and an introduction to different fuzz testing techniques. Secure development life cycle consists of security training, requirements, design, implementation, verification and release phase. Furthermore, the implementation work of the automated fuzz testing to existing testing environment and continuous integration environment is thoroughly covered. Finally, a short description of the findings is presented at the end of the thesis.

1.1 Requirements and restrictions

The goal of this thesis was to build an automated way to perform security fuzz testing for KONE proprietary protocols. This automated fuzz security testing is a part of secure Software Development Life Cycle (SDLC) procedure. Security fuzz testing was developed to be part of normal testing procedure in development phase

and to be integrated to the existing testing framework at KONE corporations IoT R&D team. The main reason behind this was to allow normal testers to have an easy way of adding fuzz testing on top of their normal testing sequences. Fuzz testing was supposed to be executed by normal testers without prior knowledge on fuzz testing. Because KONE had numerous proprietary protocols to be tested, the fuzz testing framework should be able to test all of them in various different hardware and software environments. Automated fuzz testing should be added to continuous integration development process as well. The fuzz testing should also be able to execute as a regression testing procedure. Some of the protocols are implemented in a software that runs in a very limited hardware. These protocols set restrictions the possible fuzz testing technique. All selected protocols should be able to be tested in real hardware and software environment. The selected fuzzer should be able to fuzz multiple different proprietary protocols made for different purposes. It should be able to fuzz both debug- and release builds as well.

1.2 The scope of this thesis

In this thesis the SDLC process is described to give valuable insights on how to build secure software using secure software development methods and what are the methods used to achieve this in different development phases. Since fuzz testing is effective in finding security vulnerabilities, different fuzz testing methods and fuzz testing tools are briefly described. Normal testing procedure is left out of the software testing methodology, since it is a way to validate and verify the developed product but is not a specific security testing method. Other software testing techniques that are not closely relevant to security testing are left out as well. The detailed analysis of findings is not discussed, since it is left to the developers to execute and the purpose was to develop fuzz testing. Only the fuzz testing related parts of software security testing are described in more detail. Software security testing methods suggested to be used by SDLC are discussed in different SDLC phases.

2. SECURITY DEVELOPMENT LIFE CYCLE

During the last couple of years we have seen an increasing number of different software vulnerabilities growing from hundreds to thousands as demonstrated in Figure 2.1, [1], [2]. Exploitable software has started to get the attention of press since the famous OpenSSL Heartbleed bug [3], [4]. Now software vulnerabilities are filling the headlines of IT news discussing about famous ransomwares from Petya and WannaCry to Mirai botnet [5], [6], [7], [8]. Different cyber security incidents are having more and more severe impact to the society [9]. For example, England's National Healthcare System (NHS) was heavily affected by the Wannacry ransomware causing denial of service and revenue loss [10]. To mitigate these problems, the

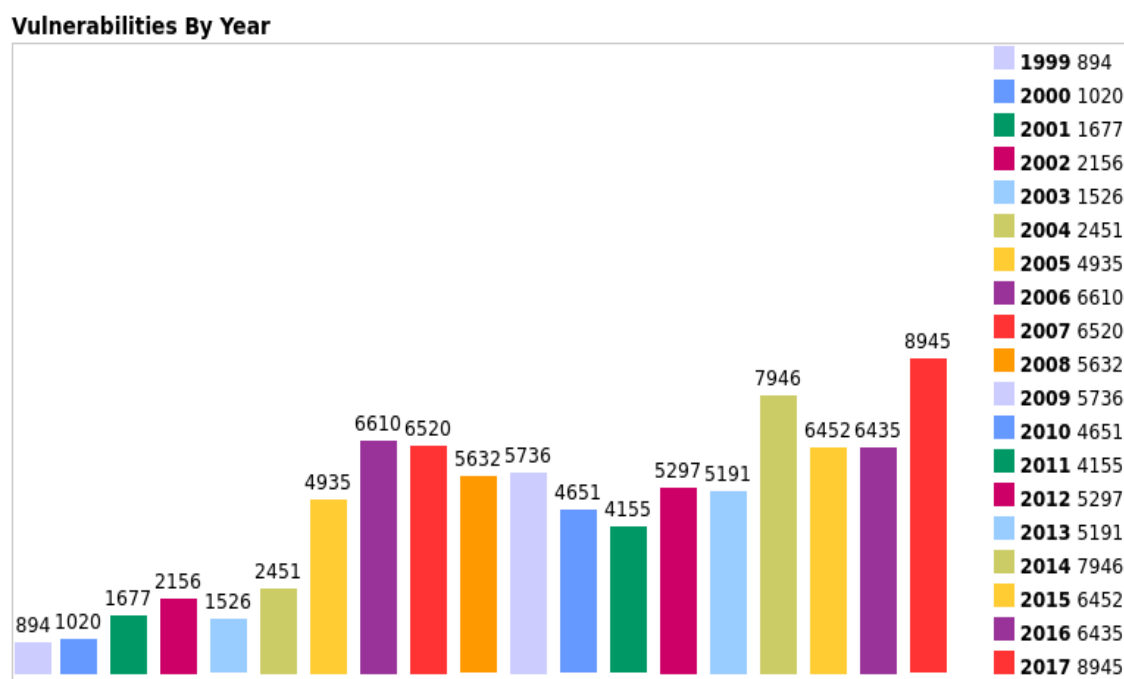


Figure 2.1 The number of reported software vulnerabilities in CVE (Common Vulnerabilities and Exposures) database over the years from 1999 to August 2017 has increased by tenfold. Figure retrieved from [1] at 8.2017.

underlying software systems should be designed by having security in mind during the entire development phase. The overall quality of the software product consist of combination of privacy, security and reliability as shown in Figure 2.2. Increasing

the security of the product will eventually increase its overall quality. Security and privacy ensures that all personal data is handled appropriately and is not easily breachable. Security and reliability ensures that the system is available all the time and can not be attacked easily with cyber attacks like DDoS (Distributed Denial of Service) [11], [12], [13], [14]. Reliability and security takes care that if the underlying system crashes no privacy related or other vice sensitive data is available in logs or error messages to unauthorized parties [15].

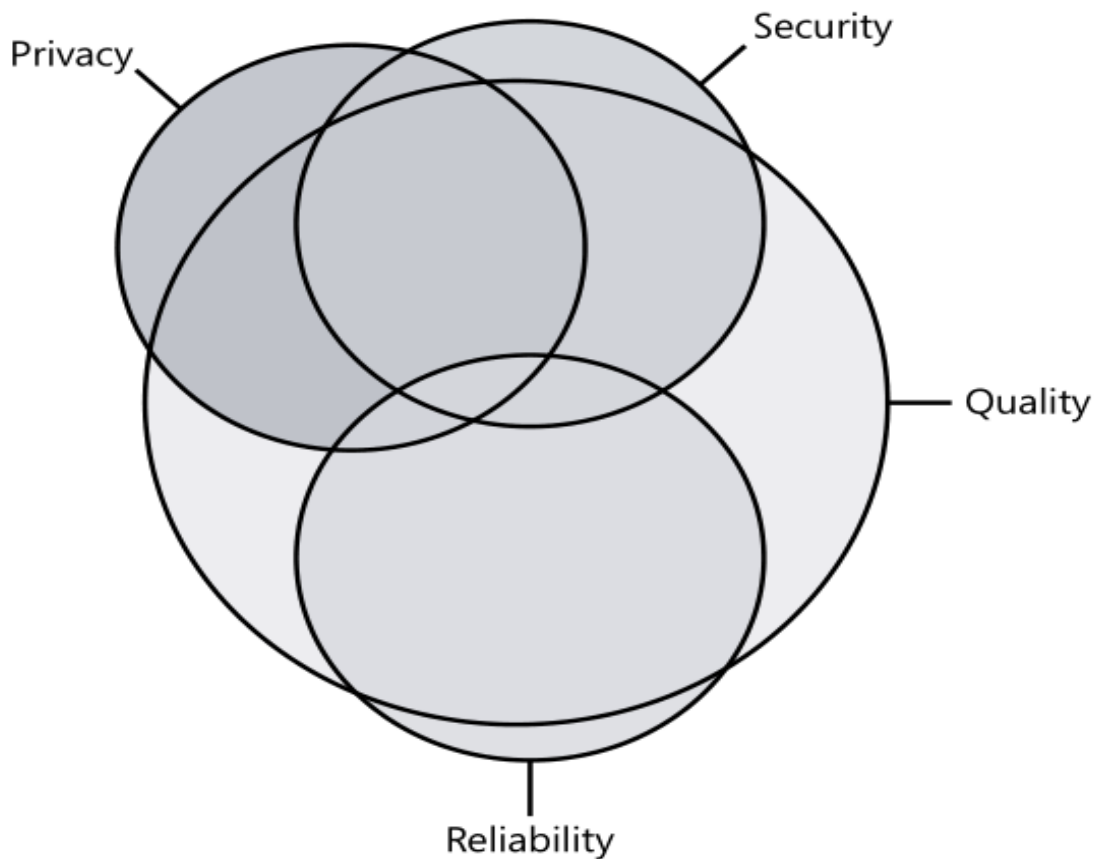


Figure 2.2 Quality of the software is a combination of security, privacy and reliability. Figure from [15].

Software security is a non-functional property of the system. It defines how a system is supposed to act or behave as well as how it should not work. Software security is part of the overall quality as illustrated in Figure 2.2 is presented. Traditionally information security has three attributes: *confidentiality*, *integrity* and *availability* (CIA). *Confidentiality* ensures that data or information is accessible only by those who are granted the access and it is not accessible by any unauthorized methods or parties. *Integrity* guarantees that there is no unauthorized or unintended modification, deletion or manipulation to data and that the data is solid. *Availability*

assures the accessibility of the data and services and denial of service is an example to violate this principle.

Faults in the design of the system can lead to flaws and implementation bugs can lead to exploitable vulnerabilities and both of these can breach the security of the system. A software vulnerability is defined by IETF (Internet Engineering Task Force): "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" [16].

There are multiple different software development methods, agile and scrum methods being to most popular ones nowadays. Many corporations adds Secure Software Development Life Cycle (SDLC) practices on top of those agile methods [17], [18]. SDLC differs from the traditional software development process by emphasizing security practices in every part of the development process. Usually this is done by having specifically defined check points and security actions to be taken in every step of the development process. This could prevent design flaws and implementation bugs from arising in the early steps in the development process. SDLC works on every resolution from architectural review to source code analysis, where one wants to find low-level programming issues. Detecting software vulnerabilities as soon as possible is known to be also more cost effective than fixing the detected issues in the future. Additionally, it has been observed that organizations that have been under severe attacks they tend to lose significant revenue and destroy a possible good reputation that they have built during the past years [19], [20].

Even though there are several different guidances, methodologies and recommendations for secure development process and testing, there is no strict definition for SDLC or for the phases that should be included. Usually, organizations or companies customizes their own guidelines for SDLC process to fit their product and organization practices [19]. SAFECode (The Software Assurance Forum for Excellence in Code) encourages to use secure design principles including threat modeling, using least privilege -method and sandboxing implementation. In addition to that, they introduce various secure coding practices for C and C++ languages. Robustness testing and fuzz testing are also mentioned in their testing recommendations [21]. Regarding the development process, SAFECode suggests to follow the best practices listed below [22]:

- Security training
- Defining security requirements
- Secure design

- Secure coding
- Secure source code handling
- Security testing
- Security documentation
- Security readiness
- Security response
- Integrity verification
- Security research
- Security evangelism

Well planned SDLC can incorporate all of those phases. Security evangelism encourages companies to publish and share good security practices.

NIST (National Institute of Standards and Technology) has provided concrete guidelines on how to incorporate security during the development process [23]. Microsoft, originally had 12 phases or "stages" in their SDLC as shown in Table 2.1.

Phase	Action
0	Education and Awareness
1	Project Inception
2	Define and Follow Design Best Practices
3	Product Risk Assessment
4	Risk Analysis
5	Creating Security Documents, Tools, and Best Practices for Customers
6	Secure Coding Policies
7	Secure Testing Policies
8	The Security Push
9	The Final Security Review
10	Security Response Planning
11	Product Release
12	Security Response Execution

Table 2.1 Microsoft used to have 12 phases in their SDLC [15].

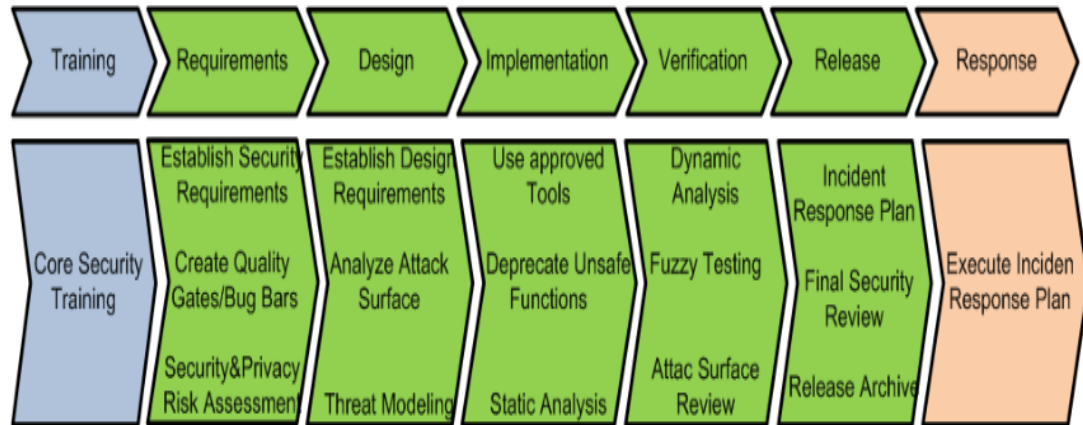


Figure 2.3 Simplified SDLC phases from Microsoft consisting of five core development phase. Training phase proceeds it and response phase in the end of it. Figure from [25].

These multiple phases has now been reduced to five core phases [24]. McAfee (former Intel Security) defines their Agile SDLC having also around five different phases quite similar to what Microsoft does [17]. Industry's best practice is to have around 5 phases in SDLC. As is shown in Figure 2.3, the core SDLC phases are: (1) requirements collection, (2) design, (3) implementation, (4) verification and (5) release. Security training as well as response are considered to be part of the SDLC process where the released product has a defined plan on how to act when security related incidents arise when the product has been released. Even though these are not considered as secure development practices, they are part of the SDLC best practices and they should be taken into consideration as well.

Although the SDLC phases differ from one publisher to another they still have certain amount of consistent phases. The new division of five phases in SDLC, as defined by Microsoft, is the one that is followed in this thesis. Those phases are discussed separately in the next chapters. Security training is a relevant part of a successful SDLC process and a brief introduction of security training practices is presented in Section 2.1. Section 1.1 discusses the requirements phase while Section 2.3 goes through the design phase of SDLC. Following the design phase, the implementation recommendations can be found in Section 2.4. Finally, the description of the verification phase is presented in Section 2.5 while the release phase can be found in Section 2.6. The rest of this thesis consists of an introduction to different fuzz testing techniques and description of the implemented fuzz testing framework.

2.1 Security Training

To successfully deliver secure software products, the underlying team should follow the best practices of secure software development. It is not guaranteed that every team member, from the architect to the test engineer, has valid security experience and education or training. It has been also observed that there is a clear lack of security awareness as well as of a formal education of security engineering. To overcome these issues, SDLC has security training as one of the required steps. Sometimes, this step is left outside the core SDLC phases. However, this does not mean that it should not be executed. The training of selected personnel can be executed in-house or as external training received from other security professionals. Various different security certificates also exist and personnel should be encouraged to complete them.

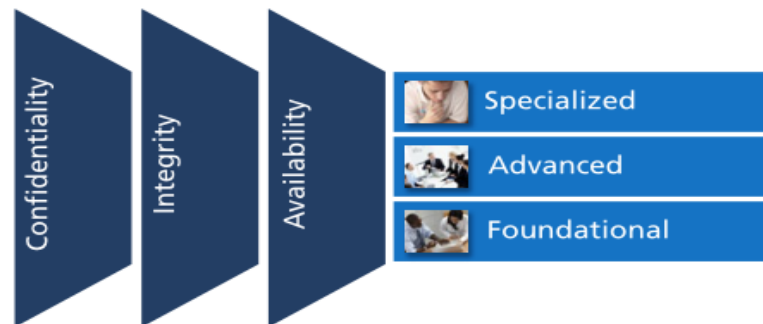


Figure 2.4 Security training can be categorized in three different levels. Figure from [26].

Currently, three main levels of security engineering training exist as shown in Figure 2.4. All of these levels, have as a main goal to increase the security awareness considering *confidentiality*, *integrity* and *availability*. The first level of security training is providing foundational information about security as it enables the security awareness. This level of knowledge is desired to exist in all members of the development teams and relevant professionals including the managers. Without appropriate security awareness among all team members it is almost impossible to build a security-aware development culture that will effectively support secure development practices.

Advanced knowledge of security engineering, includes requires developers to be aware of common secure programming practices. Covering advanced information, enables the developers and quality assurance engineers to know specific security related technical information, (e.g. programming language specific nyances). Fixing software vulnerabilities and flaws, requires a well-rounded skill set. Specialized training

should be done to engineers that seek to improve their skills up to a specialization level. Appropriate topics are for example cryptography or usage of very specific security tools. This level of security awareness is required for the professionals who work with developing security related code, like implementing the usage of cryptographic functions [26].

Secure design	Attack surface reduction
	Defense in depth
	Principle of least privilege
	Secure defaults
Threat modeling	Overview of threat modeling
	Design implications of a threat model
	Coding constraints based on a threat model
Secure coding	Buffer overruns
	Integer arithmetic errors
	Cross-site scripting
	SQL injection
Security testing	Weak cryptography
	Differences between security and functional testing
	Risk assessment
	Security testing methods
Privacy	Types of privacy-sensitive data
	Privacy design best practices
	Risk Assessment
	Privacy development best practices
	Privacy testing best practices

Table 2.2 Suggested topics for security training. Modified from [25].

The basic software security training should include the topics listed in Table 2.2, [25]. These topics, include *security design*, *threat modeling*, *secure coding*, *security testing* and *privacy*-related issues. The selected training topics depends on the underlying technologies. For example, SQL injection and cross-site scripting are only relevant if these technologies are used.

2.2 Requirements

When a software product is determined to belong under SDLC practices the secure development life cycle starts from the requirements collection phase. Sometimes,

prior to this phase the initialization phase is considered as the first step. During the initialization phase, training of the corresponding personnel as well as forming the required security team is taking place. While it is considered as a necessary step, it can be also included in the requirements phase. This is because, the needed training for personnel can be determined based on the security skills and used technologies that are required for the project. Security training and team formation is not strictly part of the SDLC practice. Thus, it is usually discussed separately from the SDLC practice phases. In this thesis, security training is discussed in Section 2.1. As shown in Figure 2.3, even though the education of personnel is not part of the core SDLC procedure it is still considered as one of the necessary steps to be followed.

In this first stage of SDLC, security requirements are defined from the business needs as well as from a business risks perspective. For defining different security requirements, the business need is the major driving force. In the requirements phase, the business need has already been realized. This, leads to the execution of risk assessment where functional parts of the system are defined and result to a concrete risk assessment documentation. Risk assessment, identifies security risks from the functional parts of the system. The identified risks, are then categorizes and ranked based on their importance. Risk assessment should be always executed in the requirements phase of SDLC and it is described in Section 2.2.2. SAFECODE states that during the requirements phase "translates the conceptual aspects of a product into a set of measurable, observable and testable requirements" [21]. US-CERT, defines risk as a "product of the probability of a threat exploiting a vulnerability and the impact to the organization" [27]. All the above definitions emphasize on how the SDLC procedure begins from the surrounding business needs and translates those requirements to a more concrete and measurable requirements and actions.

To realize and measure business needs regarding the security, a Business Impact Analysis (BIA) needs to be executed. Business Impact Analysis, helps to define critical areas of the business process and to identify the value of different security incidents for the business. Additionally, BIA is a method that helps to categorize and measure the impact of loss in confidentiality, integrity and availability in the business. Business Impact Analysis is further described in Section 2.2.1. BIA is not defined explicitly in Microsoft's SDLC but it is a method that can be useful in the requirements phase of SDLC. SQUARE (Security Quality Requirements Engineering), is another method that helps an organization to properly define security requirements in the early phase of production development. SQUARE method is analyzed in Section 2.2.3.

Different security actions are required for successfully establishing security require-

ments. Creating quality gates or bug bars [28] and security and privacy risk assessment [29], [30] are some of the recommended actions. The quality gates and bug bars are acceptance criteria. They describe what is the minimum level of security to achieve certain security goals. The idea behind bug bars is to set the accepted level for having certain bug types. For example, bugs described as critical should not be present in the released product. These are easily measurable and allows the execution of concrete actions to create a minimum acceptance level of security. Quality gate, is another easily measurable security requirement. It is similar to bug bars and usually depends on the underlying technologies. An example of a quality gate is when certain type of compiled warnings are not allowed in the release build [25].

2.2.1 Business Impact Analysis

Business Impact Analysis is a method to categorize and define the impact of the loss of confidentiality, integrity or availability in the business process and it can be used to identify critical areas in the business process. Identifying critical business process areas helps to pinpoint them for further security actions. BIA helps to identify the value of loss for the following incidents:

- Breach of confidentiality of processed data
- Loss of data integrity or process activities integrity
- Low process availability

Every found risk has a business impact that can be measured or approximated by using a process importance weight. Process importance is determined from a Business Impact Categories (BIC) Table, where BIC categories can be for example:

- Personnel or client related losses
- Law violation
- Financial losses
- Effectiveness drop

Loss level, is organization specific and the highest risk, for example, to an elevator company could be a loss of a life by using a faulty elevator or the inability to serve any personnel with the elevator causing severe personnel and client related losses.

For each found risk, the business impact value is determined. By using the Business Impact Categories Table and calculating the importance of every value with Loss Levels the process weight can be estimated. Loss levels, are ranked from lowest to the highest with a description of the possible loss. BIC Table is constructed by combining the BICs with Loss Levels. From this BIC Table, the process importance weight can be calculated with chosen metric, (e.g. by using a square sum percentage).

BIA helps defining the most important and critical business process if there is loss in data integrity, security or availability. Moreover, allows the identification of the most critical business artifacts for further security actions [31].

2.2.2 Risk Assessment

Risk assessment, identifies parts of the software that needs specific security actions and closer security inspection. Risk assessment, helps identifying parts of the project that needs to have threat modelling or secure architecture review. Other SDLC specific security testing actions like fuzz testing and penetration testing targets should also identified during the risk assessment procedure [15], [25].

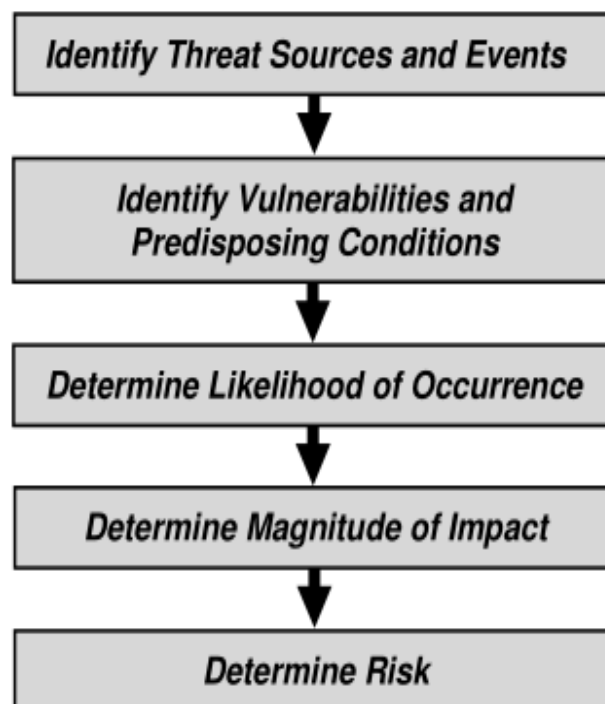


Figure 2.5 Risk assessment steps beginning of identifying threat sources and events through determining the risk magnitude and impact. Figure modified from [32].

Risk assessment can be executed at the organization level to estimate the effect of harmful events to the organizations' business, reputation and operations. It can also be executed in a lower level of operation, in certain projects and system development. Risk assessment in SDLC process estimates the impact of a breach or security incident to the developed system. Risk assessment, starts by identifying threats and threat sources that can have negative and unwanted impact to the developed system as is illustrated in Figure 2.5. Some particular events are also part of possible threats. Events that can be a threat are, for example, human error and different kinds of cyber attacks like phishing, where an attacker tries to get sensitive information hoaxing the victim. After identifying all possible risks, the impact and likelihood of the identified risks are determined. Impact, measures the severity and the effects for the organization and the system while likelihood expresses the probability of the event to occur. Risk, is the product of the likelihood of an event to happen and the impact of the event to the defined target [32].

	Security and Privacy Risk Assessment
1	Which portions of the project will require threat models before release?
2	Which portions of the project requires security design reviews before release?
3	Which portions of the project (if any) will require penetration testing by a mutually agreed upon group that is external to the project team?
4	Are there any additional testing or analysis requirements the security advisor deems necessary to mitigate security risks?
5	What is the specific scope of the fuzz testing requirements?
6	What is the Privacy Impact Rating?

Table 2.3 Security and Privacy Risk Assessment should answer these security and privacy related questions [25].

The output of a security and privacy risk assessment should provide concrete answers on the six questions that are listed in Table 2.3. All of these questions are either security or privacy related. In question number 6. is suggested to define Privacy Impact Rating. Privacy Impact Rating can be categorized as shown in Table 2.4. High privacy risk is related to products that handle, transfer or store privacy-related data. If the systems do any installation or changes file type associations, those systems are also under high privacy risk. Medium privacy risk arises from systems that do anonymous data transfer. The privacy risk by default is low (i.e. no previously described behavior exists).

Requirements of the developed product are also regulated by different laws and

High Privacy Risk	The product stores or transfers personally identifiable information changes settings or file type associations or installs software.
Moderate Privacy Risk	The sole behavior that affects privacy in the feature product, or service is a one-time, user-initiated, anonymous data transfer.
Low Privacy Risk	No behaviors exist within the product that affects privacy. No anonymous or personal data is transferred, no personally identifiable information is stored on the machine, no settings are changed on the user's behalf, no software is installed.

Table 2.4 *Privacy Impact Rating can be determined based on if the software stores or modifies personally identifiable information.*

standards [23]. For example, EU has defined the General Data Protection Regulation (GDPR) which every organization that is located in Europe needs to follow and be compliant with. In GDPR, large penalties are given if the organization does not follow the instructions considering how systematic monitoring or personal data should be handled [33]. These external requirements should be taken into account and some of them can be seen as security requirements for the developed product.

2.2.3 SQUARE

SQUARE (Security Quality Requirements Engineering) methodology was developed over 10 years ago to assist organizations to define security requirements in the early phase of development. The main goal of SQUARE, is to prevent excessive costs of fixing found product reliability and vulnerability issues during later phases of product development life cycle. SQUARE, also tries to prevent the exceeding of budget and schedule of the project, increase the quality of product and even prevent the project cancellation. This is achieved by defining all relevant stakeholders, making requirements analysis and accurate requirements specification.

Moreover, SQUARE process creates as a final product accurate and valid security requirements in 9 different steps. The process begins with agreeing on the technical definitions and outlining the core business and security goals. From these documentation other necessary artifacts are created. A method is required to claim initial security requirements from the relevant stakeholders. After deciding the method of claiming the security requirements from the stakeholders, the initial security requirements definition is executed. Last, an inspection is taking place in order to

ensure the accuracy of the defined security requirements. In Table 2.5, a detailed description of every step in SQUARE process to deliver security requirements documentation [34] is presented.

	Step	Input	Techniques	Participants	Output
1	Agree on definitions	Candidate definitions from IEEE and other standards	Structured interviews, focus group	Stakeholders, requirements team	Agreed-to definitions
2	Identify security goals	Definitions, candidate goals, business drivers, policies and procedures, examples	Facilitated work session, surveys, interviews	Stakeholders, requirements engineer	goals
3	Develop artifacts to support security requirements	Potential artifacts (e.g., scenarios, misuse cases, templates, forms)	Work session	Requirements engineer	Needed artifacts: scenario, misuse cases, models, templates, forms
4	Perform risk assessment	Misuse cases, scenarios, security goals	Risk assessment method, analysis of anticipated risk against organizational risk tolerance, including threat analysis	Requirements engineer, risk expert, stakeholders	Risk assessment results
5	Select elicitation techniques	Goals, definitions, candidate techniques, expertise of stakeholders, organizational style, culture, level of security needed, cost benefit analysis, etc.	Work session	Requirements engineer	Selected elicitation techniques
6	Elicit security requirements	Artifacts, risk assessment results, selected techniques	Joint Application Development (JAD), interviews, surveys, model-based analysis, checklists, lists of reusable requirements types, document reviews	Stakeholders facilitated by requirements engineer	Initial cut at security requirements

7	Categorize requirements as to level(system, software, etc.) and whether they are requirements or other kinds of constrains	Initial requirements, architecture	Work session using a standard set of categories	Requirements engineer, other specialist as needed	Categorize requirements
8	Prioritize requirements	Categorized requirements and risk assessment result	Prioritization methods such as Triage, Win-Win	Stakeholders facilitated by requirements engineer	Prioritized requirements
9	Requirements inspection	Prioritized requirements, candidate formal inspection technique	Inspection method such as Fagan, peer reviews	Inspection team	Initial selected requirements, documentation of decision making process and rationale

Table 2.5 All the steps in SQUARE method and the input and output of every step including the possible participants and techniques used during the step to achieve the desired output [34].

2.3 Design

The design phase of SDLC consist of secure architecture review, threat modeling, attack surface reduction and fulfilling the overall design requirements. During the design phase, the architecture of the system is created. The created architecture needs to undergo security architecture analysis to identify possible weak points, integration parts to other systems, parts of the system that needs authentication or data encryption, logging as a method of monitoring, and tracking, to mention a few. After secure architecture review the threat analysis is taking place as shown in Figure 2.6. During the threat analysis, a concrete list of attack vectors and security risks is defined. The outcome of this phase is a documentation of the architecture review and threat modeling [23].



Figure 2.6 Threat modeling is performed after architecture of the system is reviewed and it guides the following test plan [35].

2.3.1 Secure Architecture Review

Secure architecture review is an activity where the architectural design of the software product is critically reviewed. The goal is to find any weak points in the design and verify that the design fulfils both functional and non-functional security requirements. The architectural design of the system should follow the known secure design principles listed below:

- Principle of Economy of Mechanism
- Principle of Open Design
- Principle of Fail-Safe Defaults
- Principle of Least Privilege
- Principle of Least Common Mechanism
- Principle of Complete Mediation
- Principle of Separation of Privilege
- Principle of Psychological Acceptability

The designed architecture of the system should be simple and not rely on obscurity by the Principle of Economy of Mechanism. In the Principle of Open Design states, that the security mechanisms of the system should not be dependent on the secrecy of the design. The system, should be secure even though a malicious adversary could see and properly examine the overall system architecture. Furthermore, the system should have fail-safe defaults, where denying an action is the default mechanism, which is stated in the Principle of Fail-Safe Defaults. Principle of Least Privilege, indicates that operations should be executed at the lowest privilege level. Principle of Least Common Mechanics, restricts the access to resources to be private and not shared. Access to assets should be validated as is stated by the Principle of Complete

Mediation. Principle of Separation of Privilege, encourages the permission granting to be checked against multiple conditions. Last, in the Principle of Psychological Acceptability, the complexity of the system should not be affected by the underlying security mechanisms [15],[36].

Asset identification, is done on top of verifying that appropriate design principles are followed in the system architecture. During the asset identification, information assets are listed. Different assets can be cryptographic keys, database elements like user information or other valuable, and possibly private information. To verify that assets are handled securely, data flow diagrams are analyzed, if needed, to decide the needed encryption and decryption operations. Data flow diagrams can also help to determine if authentication is needed. Other actions that needs to be reviewed are listed below:

- Input validation
- Authentication
- Authorization
- Exceptions
- Integration to other systems, if any
- Auditing and Logging

In the input validation, the trust boundaries of the system are identified. When input crosses the identified trust boundary it needs to be validated. Integration to other systems can create a trust boundary and therefore those integrations needs to be examined. Actions that needs authentication and authorization needs to be visible in the system design. Any assets that are sensitive have to be stored accordingly and the appropriate cryptographic operations needs to be present for accessing and modifying the assets. Finally, the architecture needs to define which operations are going to be logged and audited. All authentication operations needs to be logged. Other actions, like data modification and configurations, should be logged as well for system monitoring [36].

2.3.2 Threat Modeling

Threat modelling is an activity where possible security threats of the system are identified and malicious behaviours are properly described (e.g. the capabilities of

a malicious adversary). The goal is to identify system threats, their impact, and design appropriate mitigations for them. The results of the threat modelling are steering the security testing plan and implemented mitigation methods. Threat modelling, should be done after the architecture of the system has been established and gone under architecture review regarding the security of the system. Threat modeling is executed based on architecture documentation, data flow diagrams, use cases, misuse cases, and other documentation that describes the underlying system and its behavior. The threat model of the system should be updated if the developed system changes during implementation [25],[37].

A team consisting of different professionals should be formed to perform the threat modelling. Even though there is no strict definition on who should be actually involved, architect and security specialists should participate in that process. Different mindsets are useful to examine threats from different views. Suitable team members should have the necessary of knowledge software security [35].

The subject of threat modelling is to list external dependencies, other vendors, identify trust boundaries, attack surfaces, and security assumptions. There are some tools available to assist threat modelling, but they can yield a lot of threats and need data flow diagrams or other description of the system as an input [38]. Security assumptions, defines which parts of the system are considered secure or trustworthy. In trust boundaries, data crosses different privilege levels and critical sections of system that handles authentication and authorization. Accessing a third party system can be identified as a crossing of the trust boundary. During the threat modelling, known weaknesses can be detected. One frequent weakness is missing authorization and it is listed in CWE (Common Weakness Enumeration) database, where the recommended prevention and mitigation actions are also listed [39],[40],[41], [35].

D amage potential	How great is the damage if the vulnerability is exploited?
R eproducibility	How easy is it to reproduce the attack?
E xploitability	How easy is it to launch an attack?
A ffected users	As a rough percentage, how many users are affected?
D iscoverability	How easy is it to find the vulnerability?

Table 2.6 DREAD security threat rating system categorizes threats and these questions helps with the threat categorization [42].

Various threat impact rating methods exist. For example, DREAD [43] can be used to determine the impact of a security threat. The letters "DREAD" can be used to answer five different questions as outlined in Table 2.6. Threats can be categorized

under DREAD and then ranked to have either *high*, *medium* or *low* impact [42]. STRIDE is another popular choice and it is introduced in Section 2.3.4. The main outcome of threat modelling is the documentation of possible threats and their mitigations.

After the threat impact is determined, the suitable mitigation methods needs to be planned. While there are different mitigation techniques, applying standard mitigations is in many cases sufficient. Attack surfaces reduction [44] is a common mitigation method and it is described in Section 2.3.3. Other common mitigation techniques are using authentication to prevent spoofing, digital signatures to assure integrity and encryption of private data to provide confidentiality. Sometimes it is not possible to apply any mitigation methods and based on the threat modelling, the architecture of the system needs to be redesigned. Lastly, the threat can be accepted as it is [35],[45].

2.3.3 Attack Surface Reduction

Attack surface is the exposure of the system to an adversary. Attack surface is a combination of multiple different access points to the system. Entry point, is a way to access the system, for example, send some input, reading from a socket or a reading from a file from a disk or network. System's exit point, is a way to transfer data from the system to the surroundings. Basically an entry point is a method that receives data and it can be either direct or indirect. A channel, is a connection to the system and eventually it invokes the system's methods. Untrusted data item of a system is an item that belongs either to an entry or exit point of the system (e.g. files and cookies). Attack surface, is a set of entry and exit points, channels and untrusted data items of the system and its environment. Obviously, the more entry and exit points exists the larger the attack surface of the system is [46].

Attack surface reduction concentrates on eliminating unnecessary items from the attack surface. These actions includes reducing the amount of code available to unauthenticated users, restricting the default privilege levels and avoid using any applications root privileges. On top of these eliminating unnecessary applications and used libraries reduces the attack surface. Limiting the used code reduces also the burden to keep it updated and the probability of different programming errors [47].

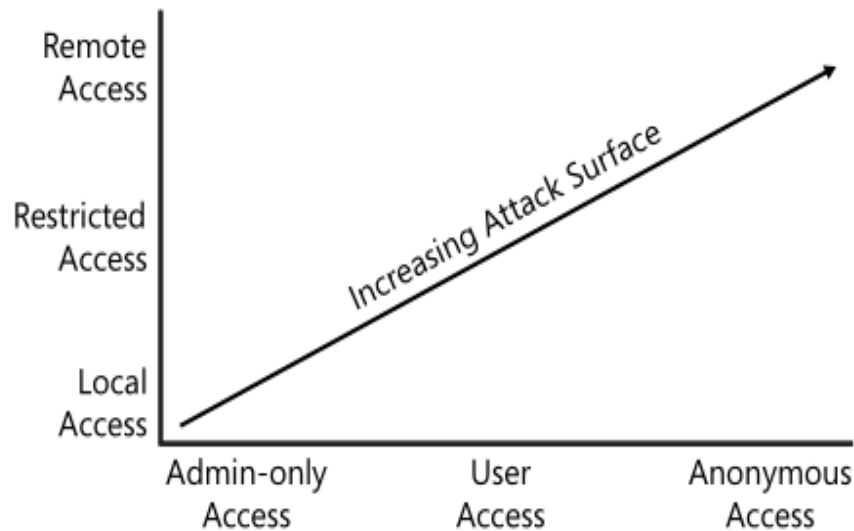


Figure 2.7 Attack surface increases when the access level type changes to more privileged. When the service is available remotely for any access it has the greatest attack surface. Figure from [15].

2.3.4 STRIDE

STRIDE outlined in Figure 2.8 is a threat taxonomy made from an attacker's perspective. It can be used to classify the found threats. When executing threat modeling the possible threats can be classified by using STRIDE method. Spoofing Identity (S) means that an attacker pretends to be someone else. Tampering (T) modifies data in a malicious way. Repudiation (R) allows an attacker to deny an action that was taken. Information Disclosure (I) reveals information to other parties that are not allowed to access it originally. Denial of Service (D) denies the usage of the system. For example Mirai botnet was a DDoS attack that prevented internet services in a large scale [5]. Elevation of Privilege (E) threat allows an attacker to gain access to more powerful privileged account, than it should be able to [15].

Spoofing Identity
Tampering
Repudiation
Information disclosure
Denial of service
Elevation of privilege

Figure 2.8 STRIDE is a threat taxonomy created by Microsoft. It can be used in threat modeling to help classify different threats from an attacker's perspective [15].

2.4 Implementation

Implementation phase consists of the actual programming and development work. During implementation phase the main purpose is to concentrate to the technical details. To achieve high quality code secure coding practices and standards should be followed. During the implementation only approved tools should be used. This includes used compilers and libraries. The followed programming standards depends on the used programming language(s). For example, when C language is used for development there is some well known functions that are not safe to use and can lead exploitable vulnerabilities. The most common memory corruptions in C or C++ language come from buffer overflow errors and unsafe buffer- and string-copying functions can cause these if extra care is not taken while programming. In Table 2.7 is an example listing of unsafe functions that should not be used and corresponding more safer functions that should be used instead. In Program 2.1 is an example of the usage of unsafe C function resulting to a buffer overflow. It is vital to prevent these type of programming errors in a secure software product.

Unsafe Function	Safe Function
strcpy	strcpy_s
strncpy	strncpy_s
strcat	strcat_s
strncat	strncat_s
scanf	scanf_s
sprintf	sprintf_s
memcpy	memcpy_s
gets	gets_s

Table 2.7 Example listing of unsafe C functions and corresponding safe C functions [21].

Good security practices includes verifying that used cryptographic algorithms are considered safe and valid. There is already outdated cryptographic algorithms that should not be used anymore, for example, MD5 and SHA1 [48]. The used cryptographic functions should use a strong entropy source for random number generation

```

1   char fromBuffer[7]={'a','b','c','d','e','f','g'};
2   char toBuffer[5]={'s','m','a','l','l'};
3   memcpy(toBuffer, fromBuffer, sizeof(fromBuffer));

```

Program 2.1 Example of a classic C buffer overflow with unsafe function *memcpy*. The function *memcpy* does not verify that the data storing buffer is large enough to save the data and thus can lead to buffer overflow.

and for cryptographic seed and nonce values. To mitigate possible programming errors manual secure code review and static code analysis should be incorporated to the development process [49], [21]. Static code analysis is discussed in Section 2.4.2 and secure code review in Section 2.4.1.

2.4.1 Secure Code Review

Secure code review is a manual code review that is executed for the product source code by a software developer or a security professional. Purpose of the secure code review is to find security related issues and it can be executed in pairs, as a team, or by an individual. The Program 2.1 illustrates a classic programming error that can be found by secure code review. Secure code review is closely related to static code analysis. The difference between these two methods is that secure code review is a manual process. Using manual code review the whole source code usually cannot be examined, like with static code analysis. For example, a static code analysis tool might not find a bug in the usage of cryptographic functions but a manual secure code review can reveal that there is a misuse of a certain function or the inputs given to the function are in wrong order or otherwise not valid. Static code analysis cannot detect if information assets are properly handled and guarded and therefore these two different source code analysis methods are complimentary [21], [37].

The high level threat model guides the secure code review process. Previously defined attack surface, threat agents, and attack vectors defined the viewpoint for the review. Threat model prioritizes the review process and narrows down the examined source code scope. Therefore examined parts of the source code depends on the existing threat model and attack vectors [37].

Secure code review examines the security related functionality. It verifies that appropriate security controls exists, secure coding principles are followed, files have correct access rights, cryptographic functions are used correctly, and that the PRNG (Pseudo Random Number Generator) is properly seeded [37]. Below is a list of some of the examination targets for secure code review:

- Logging
- Encryption and decryption
- Other cryptographic functions and methods
- Error handling

- Authentication
- Authorization
- Memory handling
- Input and output validation
- Security configuration

Logging should be carefully examined to make sure that no extra information about the system is logged and revealed to external parties. Sensitive information can leak to external parties when the software crashes and private data is leaked in to the logs. If any private data is leaked to the logs it could help an adversary to take advantage of the system. To prevent leakage of sensitive information the error handling should be executed correctly. Validating that correct security configuration is done is not enough, appropriate authentication and authorization should also be examined [37]. If a software component is provided by 3rd party, it should be analyzed for hidden functionality and for possible backdoors [50].

2.4.2 Static Code Analysis

Static code analysis tools examines the source code in an automated manner. As the name indicates, the source code is not executed and the analysis is done based only to the source code. Static code analysis is usually integrated to CI (Continuous Integration) tools to allow continuous verification of the developed source code. Since static code analysis cannot find all possible flaws it should be incorporated with secure code review practices [51]. Static code analysis tools can identify various different weaknesses. For example Juliet Test Suite which is made to evaluate static code analysis tools, covers 181 different flaws. Table 2.8 gives an example of what kind of weakness types a static code analyzing software can expose. A full listing of all possible type of issues that are able to be detected using a static code analysis would not be feasible to introduce here.

There is always a trade-off in static code analysis tools, they might introduce false positives, flagging some code to be vulnerable although it is not, and false negatives, which are failures to report vulnerable code sections. Static analysis tools have a bad reputation of suffering from high false positive rates, but during the last couple of years the quality of the tools has increased [53], [51].

authentication control	randomness
access control	error handling
buffer handling	file handling
encryption	pointer handling
reference handling	code quality

Table 2.8 *Static code analysis can expose wide range of different weaknesses. This is an example listing of possible issues that can be found by using static code analysis technique [52].*

Static code analysis tools introduces lexical analysis where the source code is parsed to distinguish function calls and variables. This kind of lexical analysis can expose calls to banned functions or libraries. While implementing the developed product it is critical to use up-to-date tools and to avoid using deprecated or banned functions and APIs (Application Programming Interface). Static code analysis can detect if these types of errors are present in the source code. Bound checking can find integer overflow and integer truncation related errors. References and pointers can introduce type confusion where incompatible pointer types are cast. C and C++ language does not introduce runtime verification for these type of errors and they can lead to data type errors. Memory allocation errors can lead to heap errors and, for example, double free, writing to already freed memory region, and buffer overflow are common mistakes resulting memory from allocation errors. These types of programming errors can be found by using source code analyzing tools [25]. The previously mentioned buffer overflow vulnerability in Program 2.1 can be detected by using static code analysis tools. The same vulnerability can be found by executing secure code review as previously mentioned in Section 2.4.1.

Data-flow analysis is used to prevent the high occurrence of false positives and negatives in static code analysis tools. It can distinguish exploitable buffer overflows from buffer overflows that can not be exploited by an attacker. Pointer-aliasing analysis is executed alongside data-flow analysis. Pointer-aliasing analysis tracks all the pointers referencing to same data location. These techniques increases the accuracy of static code analysis tools [54].

One example of a static code analysis tools is Coverity. Coverity can be integrated to existing build system and it supports various different programming languages and operating systems. Issue trackers like JIRA and Bugzilla shows Coverity results automatically. Coverity is used to scan open source projects like LibreOffice or Linux Kernel [55], [56], [57].

2.5 Verification

Last step of actual development is the verification phase where the correctness of the developed product is evaluated. If any major issues are found during the verification phase they can be still fixed before the release of the product. Microsoft suggests to perform dynamic program analysis to detect issues regarding to memory usage, different privilege levels and security problems. Dynamic analysis of the developed product includes fuzz testing. By executing fuzz testing, incorrect behavior, memory issues, and programming errors of the system can be exposed [25]. Fuzz testing is discussed more in Chapter 3.

It is common to use ready-made libraries and tools in the development and even in the end product. It would be too cumbersome to implement all the needed features in house, for example, one commonly used cryptographic toolkit is OpenSSL. Implementing all cryptographic functions in house would require excessive amount of time and resources as well as highly qualified programmers. For legal issues the used 3rd party tools licences should be reviewed and documented [17].

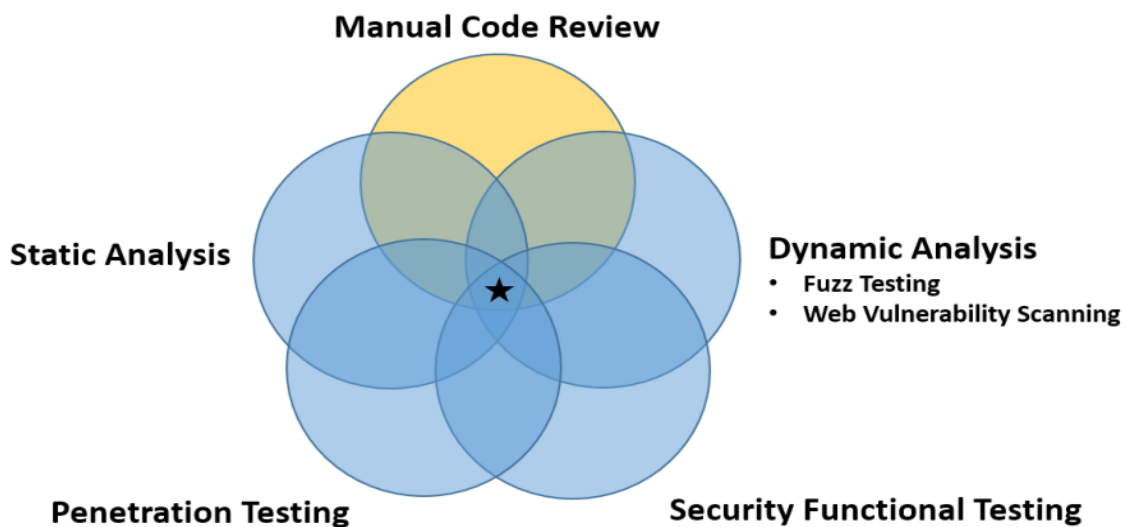


Figure 2.9 The developed source code should be tested with multiple different complementary methods to verify its correctness and robustness against security flaws and vulnerabilities. Manual code review is performed in the implementation phase [17].

Threat modeling can be executed again in the verification phase. Reason for performing threat modeling again is to detect if the original design has changed and if so, document the changes. If during the implementation phase some of the original design had to be changed, the design should be reviewed again to detect changes in attack surface. New components or a change of used technology can expose the

system to new, not previously documented, vulnerabilities. Changes in logging, authorization, or authentication should be always examined for change [25],[35].

Following the SDLC procedure the developed product should be tested with multiple complementary methods as shown in Figure 2.9. The static analysis of the source code should be executed during implementation as well as manual source code review. Later dynamic analysis testing methods like fuzz testing and, when appropriate, web vulnerability scanning are executed. Fuzzing is discussed more in detail in Chapter 3. Penetration testing is executed after other testing methods and it's performed to finished product. In the next Section 2.5.1 is a short summary of how penetration testing is done.

2.5.1 Penetration Testing

Penetration testing [58], also called pentesting, is a specific security testing technique that is performed to evaluate the security of a product. The main purpose of penetration testing is to mimic the behavior of a malicious attacker. The penetration tester tries to find exploitable design flaws and implementation bugs from the system or exploit an already known vulnerability [59],[60]. Penetration testing is usually performed as a black-box testing technique but can be also performed as a gray – or white-box testing. This is a semi-manual testing technique where automated testing tools like the network sniffer Nmap (Network Mapper) can be used on top of manual methods [61].

1.	Selecting a target PC
2.	Find target IP address
3.	Verifying the target is online
4.	Open port scanning
5.	Gaining target PC access through open ports
6.	Brute force the login credentials

Table 2.9 Hacking steps to gain access to a remote computer. Table modified from [60].

The different phases of penetration testing are presented in Figure 2.10. The testing begins from test planning and discovery of possible attack vectors, from which follows an attack phase. All the performed actions should be documented, in this way the identified vulnerabilities can be fixed. Penetration testing is a great strategy to demonstrate a system's or target's vulnerability and it is a method to discover if a system is vulnerable to different type of attacks. Example of penetration testing

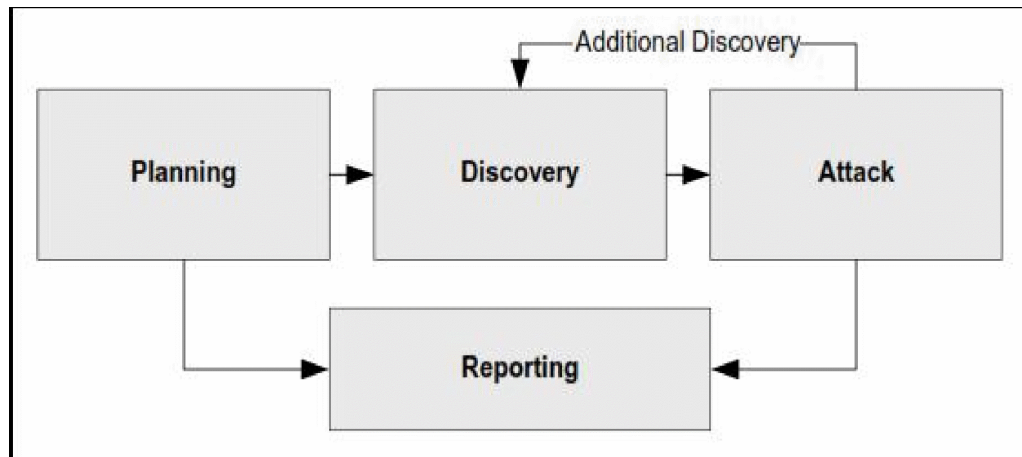


Figure 2.10 Penetration testing can be modelled having four stages starting from planning and discovery following the attack. Finally the results of testing are reported [63].

findings are to gain privilege access (root) to a system, being able to remotely execute code and gain unauthorized access to private data.

Penetration testing is a real life demonstration that can show that the system is vulnerable to certain attacks. However, fixing the identified issues is not a covered by the penetration testers and it is considered as a demanding process. Although penetration testing is suggested to be performed before the release of the software product it can be also useful to execute after the actual release [62].

2.6 Release

During the release phase, the set security requirements are revised. Prior to the release of the developed product, a process through which it is verified that that all the set and documented minimum security are fulfilled is taking place. The set security criteria (e.g. bug bars) should be met and if the set security criteria are not met the product is not considered as release-ready. At the release phase, the final security review is performed. If any security related expectations arise during the product development, they should be documented here. All expectations or deviations from the original plan should be documented and after the final review the final decision about product release and acceptance of security exceptions can be made [25]. All relevant information about the developed product needs to be saved and stored. This includes all documentation, specifications, and other product related artifacts. Storing all relevant information supports the update and maintenance of the product in the future. For the product release, an incident response plan should be made. Incident response plan describes who and what is going to be done if security related

matters arise after the release of the product. It should be planned if there is on-call contacts for possible security incidents.

Since the software ecosystem is constantly changing, the developed product needs to be updated periodically. New patches can introduce new vulnerabilities and someone can find a vulnerability that has been there for a long time but everyone else has missed it, like in the previously mentioned Heartbleed vulnerability. These kinds of issues need to be resolved and possible actions need to be properly planned.

3. FUZZ TESTING TECHNIQUES

Fuzz testing is an automated software testing technique where the test target is given modified, also called fuzzed, input. The purpose of fuzz testing, also known as fuzzing, is to find different software bugs and security related issues; vulnerabilities that can be exploited by an attacker or cause the software to malfunction. Fuzz testing can also be seen as robustness testing when the testing target SUT (System Under Test), is given multiple possibly anomalous inputs. If SUT cannot handle repeatedly given modified input values it might be easy to crash. Furthermore, fuzz testing is also a way to perform a brute force testing, also known as stress testing. This testing technique can produce endless amount of inputs for the SUT to process and monitor the SUT in several different ways [64],[65].

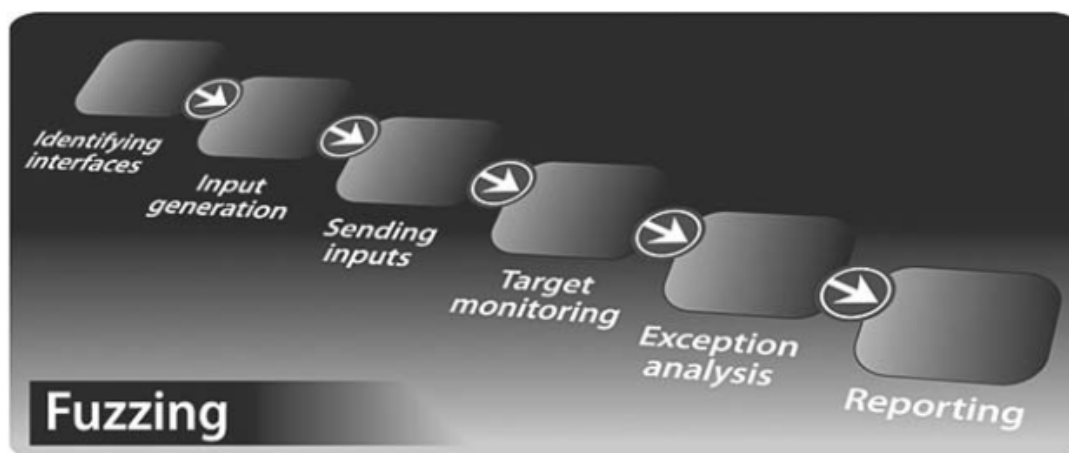


Figure 3.1 Fuzzing can be divided to phases where the fuzzing target SUT and suitable inputs are first defined. After defining the fuzzing target the fuzzed input is generated and sent to the SUT while monitoring the state of the SUT. In the last phase of fuzz testing the found crashes or exceptions are analyzed and documented [66].

Fuzz testing is usually executed during the verification phase of the SDLC as stated in Section 2.5. It can be also included in the implementation phase to detect implementation errors as soon as possible. The well-known Heartbleed bug that was mentioned in Chapter 2, has been proven to be detectable by fuzz testing [3],[67]. Fuzz testing is proved to be an effective way of finding security related vulnerabil-

ities. By repeatedly feeding the SUT malformed inputs, it tests whether the SUT is robust or not. Fuzz testing can be automated easily and can scale to test various different software systems in diverse environments and operating systems. Additionally, it is cost effective when exploitable security vulnerabilities can be found during the software development early stages. The cost of fuzz testing is relatively small, since fuzzers can generate a lot of inputs automatically and many steps of fuzz testing can be automated and scaled.



Figure 3.2 Fuzz testing can cause the underlying SUT to different failure modes. Figure modified from [66].

Fuzz testing consists of multiple different parts which can be repeated. Figure 3.1 illustrates the different fuzz testing phases. At fuzz testing, the fuzzing target is first defined. The target can be, for example, an internet browser [68],[69], a network or communication protocol [70], and different file reading programs [71]. After the target identification, the possible inputs for SUT are generated. The generation of the modified fuzzed data usually includes a valid input from which the fuzzing software starts to modify the input. The generated inputs are then sent to the SUT. Meanwhile, SUT is monitored for possible crashes and unexpected behavior. The fuzz testing target monitoring is an important phase since not all unexpected SUT behavior is easily detected. Fuzz testing can reveal that the SUT starts to use excessive amount of available resources like memory or CPU. These excessive resource usages can lead to a denial of service attack or simply a delayed response time. After the successful completion of that step, the analysis of the fuzzing findings begins. This phase, has as main goal to identify the source of the undesired behavior. Naturally, all findings are reported and documented in a concrete and strict way. Saving the fuzz testing sequence, the sent data to the

SUT, enables regression testing. Fuzz testing can reveal malfunctions in the SUT behavior. In addition to that, fuzz testing can lead to program crashes, performance degradation, and other undesired behavior as shown in Figure 3.2.

There are various different bug types that fuzz testing can successfully reveal. This varies based on the used language, technologies as well as the system environment. For example, in web programming found bug types can be related to database queries when SQL (Structured Query Language) is used. Fuzz testing, can be used to find these types of errors [72], [73]. Some common memory corruption related bug types found by fuzz testing are [66]:

- Stack overflows
- Format string errors
- Integer errors
- Heap Overflow
- (Uninitialized) Stack of Heap Variable Overwrites

A classification of different fuzz testing techniques can be done based on the information that the fuzzer has received from the testing target. Black-box fuzzers are the simplest ones having only information about the input and possible output of the SUT as illustrated in Figure 3.3. White-box fuzzer has the most knowledge of the target while gray-box fuzzers are somewhere in-between. White-box fuzzer can use the source or binary code information and have runtime tracking information of SUT by using a symbolic execution technique. Gray-box fuzzing, is a mix of both black- and white-box methods. At gray-box fuzzing the fuzzer has access to the binary code of SUT [74],[66],[64].

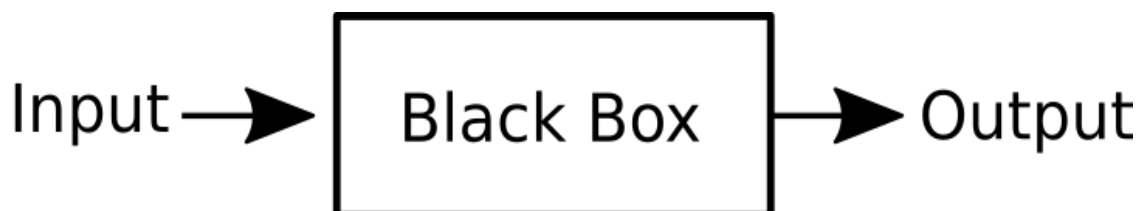


Figure 3.3 Black-box fuzzing means that the fuzzer does not have any additional information of the test target. It only receives input(s) which it uses as a seed to fuzz data for the SUT.

Other widely used division of fuzzers is the mutational-based and generation-based fuzzing. Generation-based fuzzing makes a model of the input space, which is then used to generate fuzzing inputs. Mutation-based approach does not have internal model of inputs, it uses given input samples as a seed to generate fuzzing data [64],[66].

Dividing fuzzers to strict categories is a demanding and challenging task. In this thesis, the division of fuzzer types is based on the fuzzers knowledge level of the SUT. The classification, is made to black- gray- and white-box fuzzers. Black-box fuzzers do not have any insight of the internals of the SUT while white-box fuzzers have the all information, that is, the source code of the test target. Gray-box fuzz testing lies somewhere in-between, as it can use the binary file to analyze SUT.

3.1 Black-box Fuzzing

Black-box fuzzing was the first introduced fuzzing method. It resembles random testing because it is not sophisticated in the sense that it does not have knowledge about the internal state of the SUT. Most of the black-box fuzzing tools use input as a seed to start fuzzing the data [64],[66]. There are various mutation techniques for the input. For example, the input can be modified accordingly as shown below [75]:

- Delete or introduce a random element
- Delete a sequence of things
- Repeat an element
- Duplicate a random element
- Swap two adjacent values
- Permute values

There are not many pure mutational fuzzers available. Radamsa is the most well-known black-box mutational fuzzer and it is furthered discussed in Section 4.3.2. PULSAR [76], is another black-box fuzzer to test proprietary network protocols.

Although black-box fuzz testing can be a very effective way to detect program defects it has some limitations. More precisely, it is not possible to track the source code coverage or measure execution path coverage. Advantages using this fuzzing technique are its good performance metrics. Black-box fuzz testing does not introduce

performance degradation like some white-box methods. It is also fairly simple to use because no extra compilation or instrumentation of SUT is needed [77],[78],[66].

3.1.1 PULSAR

PULSAR, has only information about the exchanged messages of the examined protocol and it models the SUT states and message structures based on that information. Furthermore, it uses reverse engineering and simulation to model protocol states and messages by using captured network traffic. PULSAR's phases are shown in Figure 3.4. In the first stage of PULSAR, a Markov model is created to represent the protocols, state machine, and message structures to model protocol messages as message templates. Rules are introduced to characterize the data flow, that is, message exchange. When PULSAR receives a certain message from SUT, it is able to categorize it to one of the created message templates and add a specific rule to describe the possible data transformation. A fuzzing mask is used to determine which parts of the message template are fuzzed. Since PULSAR has created a model of the SUT stages, it can use it to measure model coverage. Model coverage is done by introducing state machine subgraphs, where the fuzzer is guided based on how often a subgraph is visited and the amount of messages with variable input fields [76].

3.2 Gray-box Fuzzing

While black-box fuzz testing does not have any insight of the test target, white-box fuzz testing uses symbolic execution and constraints solving to explore execution paths. This, can cause significant performance degradation. Gray-box fuzz testing lies somewhere in-between, trying to get some insight of the SUT without any

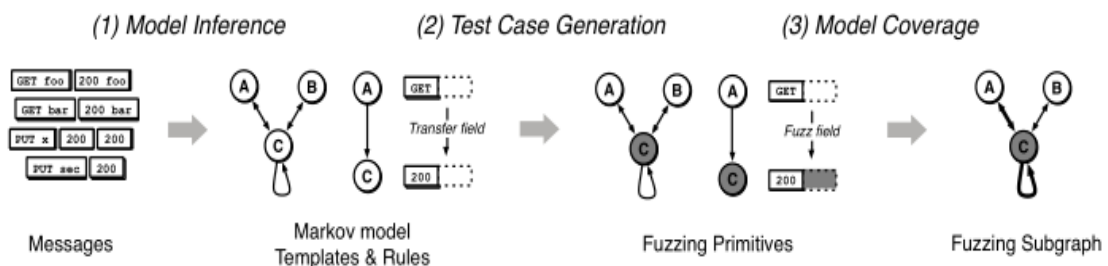


Figure 3.4 PULSAR has three (3) analysis steps: 1. Model Inference, 2. Test Case Generation and 3. Model coverage. Figure from [76].

performance cost. Lightweight binary instrumentation is used to examine program execution paths by given input without further analysis. Gray-box fuzz testing does not introduce program analysis as white-box fuzz testing does. Lightweight binary instrumentation is more scalable than deep program analysis and it can be executed in parallel [79].

A classification tree with heuristic operations method can be used for protocol fuzzing [80]. It analyzes the protocol by using a classification tree and then removes non-informational items using heuristic deductions to reduce the given input space and increase the quality of fuzzed inputs to the SUT. Heuristics operation is executed with the assistance of binary analysis program IDA Pro.

Generating the grammar describing the input data can be a cumbersome task. An automated way of creating the grammar by using machine learning was done in [81]. This approach uses an unsupervised recurrent neural network to create a statistical model of the input. This, produces a probability distribution which can be used to create novel inputs.

3.3 White-box Fuzzing

White-box fuzz testing is a fuzzing method where dynamic test generation and advanced SUT monitoring techniques are used. Symbolic execution and constraint solving is widely used in white-box fuzzing.

Limitations of white-box fuzz testing are the well known path explosion. When the size of SUT is large, it generates an endless amount of possible paths. Hence, examining all possible paths is infeasible and can lead to a small scale DDoS attack.

3.3.1 American Fuzzy Lop

American Fuzzy Lop (AFL) is an instrumented, brute-force security-oriented fuzzer. It uses compile-time instrumentation to measure edge coverage and genetic algorithms at test case discovery. Additionally, it has revealed real life vulnerabilities from various different programs like PuTTY, Mozilla Firefox, Internet Explorer and Wireshark. Whether it is a white-box fuzzer or gray-box fuzzer is a question for debate. This is due to the fact that adding compile-time instrumentation requires the software to be able to access source code but since it does not track internal state of the SUT in a precise manner and does only branch coverage it could be classified to gray-box fuzzer.

```
1 if ( x == 24 ) {  
    func1();  
3 else :  
    func2();  
5 }
```

Program 3.1 If x is 32-bit integer the else branch is executed randomly with probability of $\frac{1}{2^{32}}$ [77].

Example mutation operations used in AFL are:

- Sequential bit flip
- Boundary value substitution
- Block deletion
- Block insertion
- Arithmetic operations

AFL can be run without the compile instrumentation. For running AFL for non-instrumented binaries the QEMU mode or blind-fuzzer mode can be used. AFL is a widely used fuzzer and it has various extensions. To mention one AFL modification, [79] describes a method, named AFLFast to find low-frequency program execution paths by introducing a Markov chain model to determine that fuzzing an input seed generating a path i can also generate a path j . By adding this modified power schedule to AFL they were able to improve AFLs efficiency [82],[83].

3.3.2 SAGE

SAGE (Scalable Automated Guided Execution), is another widely known fuzzer which is able to fuzz Windows file-reading applications. It uses symbolic execution and code coverage maximizing heuristics to improve test case generation and vulnerability detection. Since black-box fuzzing has problems to cover if-else -type branching by randomly mutating the input value as shown in Figure 3.1, symbolic execution with path constraints is used to overcome these type of code coverage issues. In SAGE, test cases are created dynamically by analyzing path constraints using a negation technique. Coverage-maximizing search algorithm is used to increase vulnerability detection performance. Furthermore, SAGE uses generational search to mitigate the path explosion problem [77].

White-box fuzzing suffers from the path explosion problem, where all the possible program execution paths lead to an infeasible amount of possible paths to explore. In [84] authors introduced a white-box fuzz testing technique which does grammar-based analysis for the valid inputs to enable deeper path coverage. Greater path coverage is achieved by enabling the fuzzer to generate always parseable inputs. Furthermore, a novel dynamic test generation algorithm was introduced. This algorithm, generates grammar-based constraints in symbolic execution with a constraint solver. This grammar-based white-box fuzzer performed better than black-box, white-box or only grammar-based fuzzers when it was executed against Internet Explorer 7 JavaScript interpreter.

4. IMPLEMENTATION OF THE FUZZ TESTING FRAMEWORK

The main contribution of this work, is the development of a fuzz testing framework. This chapter, describes the implementation of the developed fuzz testing framework. This fuzz testing framework enables fuzz testing to be executed in SDLC implementation phase as well as in SDLC verification phase. Furthermore, the fuzz testing framework is able to fuzz the developed protocols in an automated manner by integrating fuzzing to CI pipeline and by executing selected tests automatically. As having fuzzing included in CI pipeline, testers are able to execute short term fuzz tests as well as long term fuzz testing. If any of the executed fuzz tests fails the produced fuzzed packets and fuzz testing sequence can be used at regression testing, and as an acceptance criteria for the developed product. Since the fuzz testing is automated, longer fuzz tests can be executed during the verification phase of the development if it is required. Enabling fuzz testing in multiple parts of SDLC it is considered as a cost effective technique since we can start fuzz testing the target protocols in the early phase of development as well as later – before the release of the product.

To be able to access the desired protocol packets testing framework captured the original protocol packets and modified them as needed. Robot Framework was used to derive the SUT to desired state to capture the desired protocol messages if needed. Automatic reporting of test cases was also done with the Robot Framework [85]. Robot Framework is described in details in Section 4.1.1. Accessing a specific protocol packet was achieved by having a MITM style approach where the fuzzing framework received the protocol packet from the testing framework and then returned the modified protocol packet back to the testing framework. By returning the modified protocol packet it be could processed inside correct protocol packets, for example, add appropriate TCP and IP protocol headers automatically by the existing testing framework. In Figure 4.1 is a visualization how fuzz testing framework has access to the selected protocol packet and how the fuzz testing framework can return the processed protocol to the testing framework which can build the rest of the protocol stack correctly.

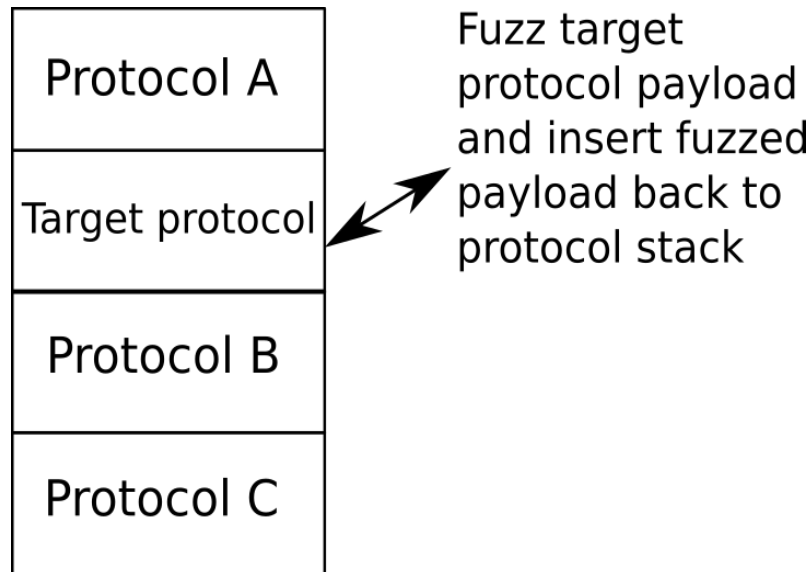


Figure 4.1 Fuzzing framework is able to fuzz separate protocol packets by mimicking a MITM behavior where it can access the protocol stack, modify a desired protocol packet and return the modified protocol packet back to the testing framework before it is sent forward to the SUT.

4.1 Automated Software Testing tools

Automated software testing tools were used to enable more accurate and precise testing of target protocols. Since the proprietary protocol stack might be complex and not all protocols were tested simultaneously an automated testing tool was used to drive the testing target to a desired state. This action enables the testing of a specific protocol and to bypass other possible protocols in the protocol stack. Using an automated testing tool enables also automated log collection and reporting of the progress of test cases.

4.1.1 Robot Framework

Robot Framework is an open source test automation framework that can be used to execute data driven tests and acceptance testing. It is operating system independent and developed in Python. Robot Framework has keyword driven tests and Python or Java implemented libraries can be used to expand its testing capabilities. The test cases should be designed to be human readable. Human readable test cases are achieved by adding layers of different keywords to hide the technical details [85].

Robot Framework is used in KONE testing environment. Robot Framework can be used to drive SUT to a desired state. By introducing SUT some state controlling,

fuzzing can be executed to, for example, a connected protocol state to overcome exhaustive fuzz testing against closed connection state. SUT state controlling also enables testers to fuzz test selected protocol messages. This enable fuzz testing a new developed feature early in the development phase to detect possible implementation bugs as soon as possible. It also enables that all the desired protocol messages are fuzz tested.

Robot Framework is able to launch the testing target, if needed, initialize execution environment and provide the needed test files. This can be achieved using the Suite Setup and Test Setup keywords. The corresponding cleaning up -functionality is achieved with Suite- and Test Teardown keywords. Robot Framework is able to collect all the test case related logs to a certain location and it provides easy interface to examine the test execution. These SUT controlling steps are vital, for example, some hardware environments could accept corrupted configurations and after a reboot cause undesired behavior to the device. This is prevented by using the Suite- and Test Teardown functionality ensuring that the SUT is left in a good state after fuzz testing.

4.2 Continuous Integration tools

Continuous integration tools are software that are used to automate software building and testing steps, for example. These can be integrated to a software version control systems, like git and svn, and to trigger software building. CI can also execute source code analysis tools. These tools can be executed automatically to examine the quality of committed source code and to alert developers if any issues arise. Static code analysis is usually executed in CI tooling. Automated test case execution is also a part of the build process. After building the new software version the selected tests are executed to ensure that the new version of the software passes the tests. The used CI tool is discussed in the next Section 4.2.1.

The trend nowadays is to use CI in the implementation. CI enables source code inspection and building in real time resulting in a continuous delivery of a software product. Many of the used secure development practices can be added to a CI, for example, normal testing methods and static code analysis can be automated. Adding fuzz testing on top of normal testing can capture implementation bugs in an early phase and reveal unexpected faults. Including fuzz testing expands the test space from normal test cases and defined bad test cases to arbitrary test case space as can be seen in Figure 4.2.

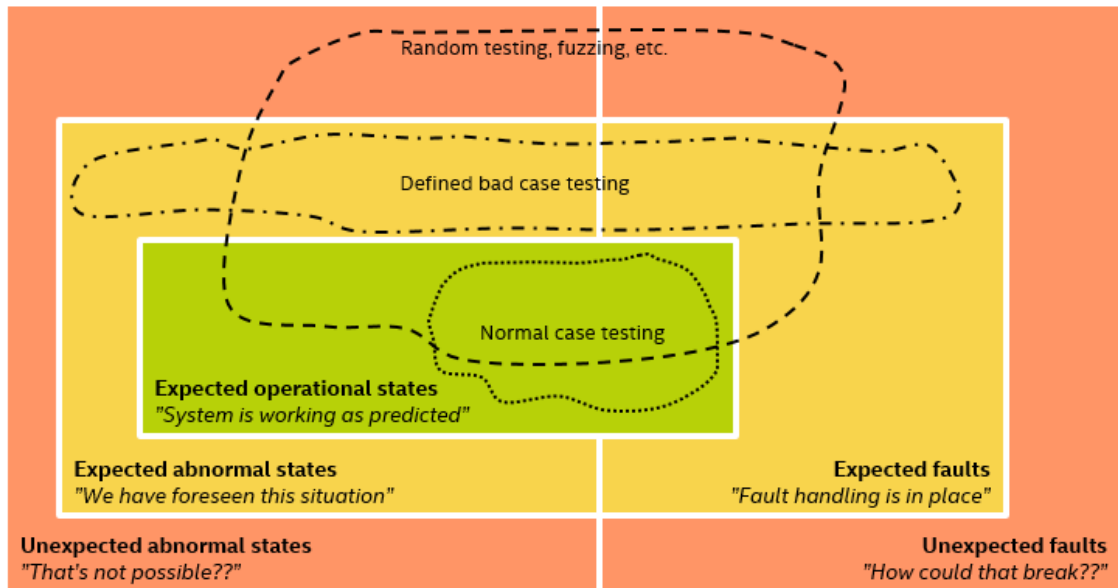


Figure 4.2 Fuzz testing can capture the states of defined bad cases as well as normal cases. Beyond those fuzz testing can capture unexpected abnormal states and unexpected faults, which are not captured by defining bad test cases or normal test cases [86].

4.2.1 Jenkins

The developed fuzz testing framework was included in a continuous integration tool. Integration to CI tools allows the automatic execution of fuzz test cases. Automatic test execution can be used to execute also regression testing and acceptance testing. Regression testing is used to verify that founded issues are fixed properly. Acceptance testing is a longer time period testing where the SUT is fuzz tested for selected time period. The selected CI tool to achieve this functionality was Jenkins. Jenkins is Java based automation server. It can automated pre-defined tasks, it can fetch source code and build it and execute tests among other various possibilities [87]. To execute fuzz testing against selected targets in an automated manner a Jenkins server was installed. In Figure 4.3 is a presentation of how CI enables developers and testers get feedback by automatically executing test cases. Multiple fuzzing execution tasks for Jenkins was built. The goal of these tasks was to update the tested source code automatically and run selected long and short term fuzz tests for different target protocols. Finally the possible findings can be reported automatically.

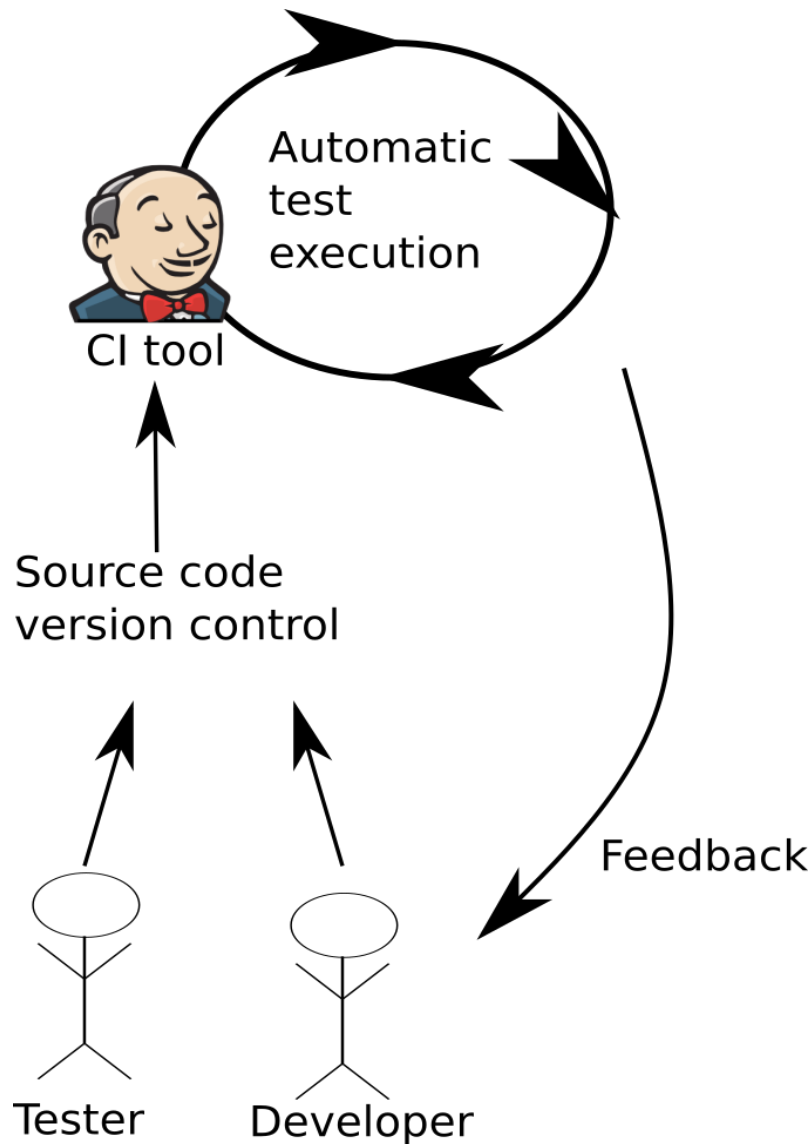


Figure 4.3 Continuous integration enables the testers and developers to get instant feedback of their work by executing different tests automatically.

4.3 Used Fuzz Testing tool

The fuzz testing at KONE IoT R&D is executed in varying hardware and software environment. KONE has various different hardware products and the lifecycle of the products is especially long and therefore creates challenging environment. This restricts the use of some white box fuzz testing tools, since it is desired that the selected testing targets are able to be tested in real hardware environment. In the Section 4.3.1 the requirements for the selected fuzz testing tool are listed. These requirements arise from the existing KONE software development and testing environment. In the Section 4.3.2 the selected fuzz testing tool is described.

4.3.1 Fuzz Testing Tool Requirements

Fuzz testing was developed to be a part of normal testing procedure and to be integrated to the existing testing framework at KONE Corporation's IoT R&D section. This was done to enable software testers to have easy way of adding fuzz testing on top of their normal testing sequences. The fuzz testing was supposed to be able to be executed by testers without prior knowledge about fuzz testing. Because KONE has numerous proprietary protocols to be tested, the fuzz testing framework should be able to test all of them in various different hardware and software environments. Automated fuzz testing should be added to continuous integration development process as well. The fuzz testing should also be able to execute as regression testing procedure. Some of the protocols are implemented in software that runs in a very limited hardware. This restricts the execution environment since it could be difficult to add any modified binaries to the testing target. To be able to test as many protocols as possible it is easiest to develop a framework that is using a black-box fuzzer. Below is a summary of the restrictions and requirements for the selected fuzz testing tool.

1. Needs to have found real life vulnerabilities
2. Is actively maintained
3. Able to test various protocols in real hardware and software environment
4. Able to test both debug- and release builds
5. Able to be integrated to continuous integration tool
6. Able to be integrated to existing KONE testing environment
7. Fuzzing sequence can be saved for regression testing
8. Does not require the source code (black-box fuzzing)

4.3.2 Radamsa Fuzzer

Radamsa is a well known black-box fuzzer and general purpose fuzzer. It is developed initially at University of Oulu at Oulu University Secure Programming Group (OUSPG). Radamsa is described to be a "state of the art black-box robustness testing" software. The authors describe that Radamsa can be added to SDLC process to increase the overall quality of the developed system. Radamsa is a lightweight

tested only after proper authentication, while some of the protocols are only able to be tested at the target hardware. The variety of fuzz test targets obligates to make more fine grained testing plan for each of the tested protocol. The authentication restriction is somewhat easy to bypass, by driving the SUT to the desired testing state by making the authentication first, but the target testing is causing more difficulties, like SUT monitoring problems, especially when a product build is used. The SUT is hard to monitor when debuggin build features, like remote SSH (Secure Shell) connection, is not present. The SUT should be monitored for excessive CPU, memory and other resource usage and for slow responses. For monitoring these resource usages and SUT logging, while using a production build, is not a trivial task. Other proprietary KONE protocol was used to help overcome some of these target monitoring issues.

After these KONE specific impediments, the common fuzz testing issues are present; how to reproduce the found issues? It is not always easy to reproduce the examined SUT behaviour. Sometimes we are not able to reproduce the same behaviour with the same data and the found issue is flagged as false positive. Also defining which of the sent fuzzed data is causing the examined behaviour in SUT is not easy, especially when the fuzz testing has been ongoing for a long period and the SUT has parsed large amount of fuzzed data. What is common with the tested protocols is to close the connection if the sent data to the SUT is not valid. When the connection is closed by the SUT quite often, it is difficult to sent a lot of fuzzed data to the target. This type of issue causes the fuzz testing to be more of a "open a connection, send couple of packets of fuzzed data and get connection closed message" -type of testing sequence. In the end, we end up reopening the connection several times, without being able to sent multiple fuzzed protocol packets. This is a good sign, because the SUT is able to distinguish the fuzzed data from normal valid data, but it also makes it harder to fuzz the SUT thoroughly.

4.5 Fuzz Testing Findings

A fuzz testing framework was developed and integrated to a CI server. Some selected proprietary protocols were fuzz tested using Radamsa fuzzer and the developed fuzz testing framework. The developed system was successful in finding several implementation errors. As mentioned in Section 1.2, detailed analysis of the fuzz test findings was out of the scope of this work. As such only a short summary of the findings is listed. Table 4.1 lists some of the found issues during fuzz testing. The goal here was to find different security-related protocol implementation vulnerabilities. As stated in Table 4.1 input validation errors occurred and those findings can

lead to several security-related vulnerabilities. A detailed analysis of the root cause of the findings was left to the developers to examine and it is outside of the scope of this work.

Fuzz testing findings	
1.	Input validation errors
2.	No forced bound checking
3.	Wrong implementation of variable type
4.	No proper handling of empty input values
5.	Improper bitmask implementation
6.	Excessive usage of memory

Table 4.1 Example findings of implemented fuzz testing framework.

5. POSSIBLE FUTURE WORK

In this thesis a fuzz testing framework was designed and implemented. The developed fuzz testing framework was integrated to an existing automated testing framework and continuous integration system. Integration to the existing automated testing framework enables to fuzz test desired KONE proprietary protocols and specific packets or messages of the protocol. The implemented fuzzing method was black-box fuzzing. In this fuzz testing framework implementation, Radamsa fuzzer was used as a black-box fuzzing tool.

In the future, some other black-box fuzzer could be used to investigate if more implementation bugs and vulnerabilities can be found by fuzz testing. For example, AFL fuzzer can be configured to a black-box fuzzer and it could be used as the fuzzer. Other fuzz testing strategy like white-box fuzzing could be implemented in the future to enable even more thorough fuzzing. By enabling the fuzzer to have some insight of the protocol state and fuzz testing coverage a more thorough fuzz testing could be achieved. Since the designed protocol specifications and source code are available, white-box fuzzing could be used to fuzz test the protocol implementations. A white-box fuzz testing would be able to benefit from the source code or binary and include coverage measurements for the fuzzing. By adding coverage measurement the target protocol implementation would be fuzz tested more thoroughly. The protocol implementations could be fuzz tested at a host PC, if fuzz testing at the dedicated hardware is not possible due the hardware restrictions. A template based fuzzing approach could be beneficial as well, it would take into account the protocol fields. A delicate error or change in the protocol packet could be more efficient to find security vulnerabilities. For those protocols that do not have a payload, a state machine fuzzing technique could be implemented to detect abnormal behavior of the SUT. For example, if the protocol is in closed connection state the state machine fuzzing could try to change the state arbitrarily from closed to something different.

A commercial fuzzer might be good solution to identify more issues during fuzz testing. Defensics fuzzer has Software Development Kit (SDK) to define custom protocols. With the Defensics SDK Framework, the protocol structure and the state machine of the protocol can be properly defined. This would allow more

specific fuzzing strategies to be used, while fuzz testing. The main drawback of this approach is the manual work that is required in order to define the protocols as well as their state machines. Thus, this type of solution would suit a target protocol that is otherwise hard to fuzz test or is easy to implement with the Defensics SDK fuzz testing framework. It is obvious, that when the amount of protocols to fuzz test is quite large and the type of the protocols varies no perfect solution, that would work on all of these protocols, exists. By dividing the target protocol types and doing other type of classification of the protocols we can narrow down the number of optimal approaches to consider for each of these protocol sets.

6. CONCLUSIONS

The purpose of this thesis was to create a fuzz testing framework that is incorporated to the existing testing and continuous integration environment at KONE corporation. The goal was to enable fuzz testing in SDLC implementation and verification phases. The fuzz testing framework is included in a continuous integration pipeline. The test target was multiple different KONE proprietary protocols. The existing KONE testing and development environment was developed by using Jenkins continuous integration server and Robot Framework testing corpus. Furthermore, the developed fuzz testing framework was integrated to these tools. In the fuzz testing framework, a black-box fuzzer Radamsa was used. The built environment was successful in finding security related vulnerabilities from the selected protocols that were tested. The fuzz testing environment enables regression testing to verify that the founded issues are fixed correctly to prevent them showing up again. Finally, the built fuzz testing framework was made easy to use for software testers.

BIBLIOGRAPHY

- [1] “CVE details: Vulnerabilities by date,” August 2017, accessed: 9.8.2017. [Online]. Available: <http://www.cvedetails.com/browse-by-date.php>
- [2] Y. Younan, “25 Years of Vulnerabilities: 1988-2012,” March 2013, accessed: 9.8.2017. [Online]. Available: <https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf>
- [3] Codenomicon, “The Heartbleed Bug,” April 2014, accessed: 29.7.2017. [Online]. Available: <http://heartbleed.com/>
- [4] “CVE-2014-0160,” April 2014, accessed: 5.8.2017. [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160>
- [5] N. Perlroth, “Hackers Used New Weapons to Disrupt Major Websites Across U.S.” October 2016, accessed: 9.8.2017. [Online]. Available: https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html?hp&action=click&pgtype=Homepage&clickSource=story-heading&module=first-column-region®ion=top-news&WT.nav=top-news&_r=0
- [6] L. H. Newman, “The Botnet That Broke the Internet Isn’t Going Away,” December 2016, accessed: 9.8.2017. [Online]. Available: <https://www.wired.com/2016/12/botnet-broke-internet-isnt-going-away/>
- [7] M. H. Matti Konttinen, “Maailmalla leviävä kiristysohjelma on rantautunut myös Suomeen - "Ei tule pysähtymään nyt eikä koskaan",” June 2017, accessed: 9.8.2017. [Online]. Available: <https://yle.fi/uutiset/3-9693275>
- [8] GReAT, “Wannacry ransomware used in widespread attacks all over the world,” May 2017, accessed: 9.8.2017. [Online]. Available: <https://securelist.com/wannacry-ransomware-used-in-widespread-attacks-all-over-the-world/78351/>
- [9] N. B. Alexander Smith, Saphora Smith and P. Cahill, “Why ”WannaCry” Malware Caused Chaos for National Health Service in U.K.” May 2017, accessed: 12.8.2017. [Online]. Available: <http://www.nbcnews.com/news/world/why-wannacry-malware-caused-chaos-national-health-service-u-k-n760126>
- [10] F-Secure Labs, “WannaCry, the Biggest Ransomware Outbreak Ever,” May 2017, accessed: 9.8.2017. [Online]. Available: <https://safeandsavvy.f-secure.com/2017/05/12/wannacry-may-be-the-biggest-cyber-outbreak-since-conficker/>

- [11] A. Michalas, N. Komninos, N. R. Prasad, and V. A. Oleshchuk, “New client puzzle approach for dos resistance in ad hoc networks,” in *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference*. IEEE, 2010, pp. 568–573.
- [12] A. Michalas, N. Komninos, and N. R. Prasad, “Mitigate DoS and DDoS attack in mobile ad hoc networks,” *International Journal of Digital Crime and Forensics (IJDCF)*, vol. 3, no. 1, pp. 14–36, 2011.
- [13] A. Michalas, N. Komninos, and N. Prasad, “Multiplayer game for ddos attacks resilience in ad hoc networks,” in *Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, February 2011, pp. 1–5.
- [14] A. Michalas, N. Komninos, and N. R. Prasad, “Cryptographic puzzles and game theory against dos and ddos attacks in networks,” *International Journal of Computer Research*, vol. 19, no. 1, p. 79, 2012.
- [15] M. Howard and S. Lipner, *The Security Development Lifecycle. SDL: A Process for Developing Demonstrably More Secure Software*, 1st ed. Microsoft Corporation, May 2006.
- [16] R. Shirey, “Internet Security Glossary, Version 2,” August 2007, accessed: 9.8.2017. [Online]. Available: <https://tools.ietf.org/html/rfc4949>
- [17] McAfee, “McAfee Product Security Practices,” April 2017, accessed: 18.7.2017. [Online]. Available: <https://www.mcafee.com/us/resources/misc/ms-product-software-security-practices.pdf>
- [18] C. Pohl and H. Hof, “Secure Scrum: Development of Secure Software with Scrum,” *CoRR*, vol. abs/1507.02992, 2015. [Online]. Available: <http://arxiv.org/abs/1507.02992>
- [19] M. Kara, “Review on common criteria as a secure software development model,” *International Journal of Computer Science & Information Technology*, vol. 4, no. 2, p. 83, 2012.
- [20] NIST, “NIST Special Publication 800-37r1, Guide for Applying the Risk Management Framework to Federal Information Systems: A Security Life Cycle Approach,” February 2010, accessed: 10.11.2017. [Online]. Available: <http://dx.doi.org/10.6028/NIST.SP.800-37r1>

- [21] M. Belk, M. Coles, C. Goldschmidt, M. Howard, M. Saario, R. Sondhi, I. Tarandach, A. Vähä-Sipilä, and Y. Yonchev, “Principles for Software Assurance Assessment. A Framework for Examining the Secure Development Processes of Commercial Technology Providers,” February 2011, accessed: 30.7.2017. [Online]. Available: http://www.safecode.org/wp-content/uploads/2014/09/SAFECode_Dev_Practices0211.pdf
- [22] SAFECode, “Software Assurance: An Overview of Current Industry Best Practices,” February 2008, accessed: 9.8.2017. [Online]. Available: http://www.safecode.org/publication/SAFECode_BestPractices0208.pdf
- [23] R. L. Kissel, K. M. Stine, M. A. Scholl, H. Rossman, J. Fahlsing, and J. Gulick, “Security Considerations in the System Development Life Cycle,” October 2008, accessed: 12.8.2017. [Online]. Available: <https://www.nist.gov/publications/security-considerations-system-development-life-cycle>
- [24] Microsoft, “Microsoft Security Development Lifecycle (SDL) version 5.2,” 2012, accessed: 29.7.2017. [Online]. Available: <https://msdn.microsoft.com/en-us/library/windows/desktop/cc307748.aspx>
- [25] Microsoft, “Microsoft Security Development Lifecycle (SDL),” 2012, accessed: 16.9.2017. [Online]. Available: <https://www.microsoft.com/en-us/SDL/>
- [26] SAFECode, “Security Engineering Training - A framework for Corporate Training Programs on the Principles of Secure Software Development,” April 2009, accessed: 30.7.2017. [Online]. Available: http://www.safecode.org/publication/SAFECode_Training0409.pdf
- [27] US-CERT, “Architectural Risk Analysis,” July 2013, accessed: 11.8.2017. [Online]. Available: <https://www.us-cert.gov/bsi/articles/best-practices/architectural-risk-analysis/architectural-risk-analysis>
- [28] G. Schermann, J. Cito, P. Leitner, and H. C. Gall, “Towards quality gates in continuous delivery and deployment,” in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–4.
- [29] T. Dimitriou and A. Michalas, “Multi-Party Trust Computation in Decentralized Environments,” in *2012 5th International Conference on New Technologies, Mobility and Security (NTMS)*, May 2012, pp. 1–5.
- [30] T. Dimitriou and A. Michalas, “Multi-party Trust Computation in Decentralized Environments in the Presence of Malicious Adversaries,” *Ad Hoc Networks*, vol. 15, pp. 53–66, April 2014. [Online]. Available: <http://dx.doi.org/10.1016/j.adhoc.2013.04.013>

- [31] J. Baginski and M. Rostanski, “The Modeling of Business Impact Analysis for the Loss of Integrity, Confidentiality and Availability in Business Processes and Data,” *Theoretical and Applied Informatics*, vol. 23, no. 1, p. 73, 2011.
- [32] J. T. F. T. Initiative, “Guide for Conducting Risk Assessments,” September 2012, accessed: 12.8.2017. [Online]. Available: <https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final>
- [33] “The EU General Data Protection Regulation,” 2016, accessed: 29.7.2017. [Online]. Available: <http://www.eugdpr.org/>
- [34] W. D. Yu and K. Le, “Towards a Secure Software Development Lifecycle with SQUARE+R,” in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, July 2012, pp. 565–570.
- [35] SAFECode, “Tactical Thread Modeling,” 2017, accessed: 30.7.2017. [Online]. Available: https://www.safecode.org/wp-content/uploads/2017/05/SAFECode_TM_Whitepaper.pdf
- [36] P. H. Meland and J. Jensen, “Secure Software Design in Practice,” in *2008 Third International Conference on Availability, Reliability and Security*, March 2008, pp. 1164–1171.
- [37] OWASP, “OWASP Code Review Guide,” 2008, accessed: 2.12.2017. [Online]. Available: https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf
- [38] C. Möckel and A. E. Abdallah, “Threat modeling approaches and tools for securing architectural designs of an e-banking application,” in *2010 Sixth International Conference on Information Assurance and Security*, August 2010, pp. 149–154.
- [39] MITRE, “Common Weakness Enumeration,” accessed: 10.1.2017. [Online]. Available: <http://cwe.mitre.org/>
- [40] “CWE-862: Missing Authorization,” accessed: 10.1.2017. [Online]. Available: <http://cwe.mitre.org/top25/#CWE-862>
- [41] D. Dhillon, “Developer-driven threat modeling: Lessons learned in the trenches,” *IEEE Security Privacy*, vol. 9, no. 4, pp. 41–47, July 2011.
- [42] Microsoft, “Thread Modeling,” 2003, accessed: 6.12.2017. [Online]. Available: <https://msdn.microsoft.com/en-us/library/ff648644.aspx>

- [43] R. Gregory and R. Mendelsohn, "Perceived Risk, Dread, and Benefits," *Risk Analysis*, vol. 13, no. 3, pp. 259–264, 1993. [Online]. Available: <http://dx.doi.org/10.1111/j.1539-6924.1993.tb01077.x>
- [44] D. Kantola, E. Chin, W. He, and D. Wagner, "Reducing Attack Surfaces for Intra-application Communication in Android," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 69–80. [Online]. Available: <http://doi.acm.org/10.1145/2381934.2381948>
- [45] Microsoft, "Introduction to Threat Modeling," accessed: 16.9.2017. [Online]. Available: https://download.microsoft.com/download/9/3/5/935520EC-D9E2-413E-BEA7-0B865A79B18C/Introduction_to_Threat_Modeling.ppsx
- [46] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, May 2011.
- [47] D. Colesniuc and I. Martin, "Cybersecurity By Minimizing Attack Surfaces," in *International Scientific Conference "Strategies XXI"*, vol. 1. "Carol I" National Defence University, 2015, pp. 42–48.
- [48] NIST, "NIST Policy on Hash Functions," August 2015, accessed: 19.1.2018. [Online]. Available: <https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions>
- [49] Intel, "safestringlib," March 2017, accessed: 18.7.2017. [Online]. Available: <https://github.com/01org/safestringlib>
- [50] C. Wysopal and C. Eng, "Static detection of application backdoors," August 2007, accessed: 2.12.2017. [Online]. Available: <https://www.veracode.com/blog/2007/08/blackhat-2007-materials>
- [51] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, no. Supplement C, pp. 18 – 33, 2015. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0950584915001366>
- [52] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, vol. 45, no. 10, pp. 88–90, October 2012.
- [53] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code," *SIGSOFT*

- Softw. Eng. Notes*, vol. 29, no. 6, pp. 97–106, 2004. [Online]. Available: <http://doi.acm.org/10.1145/1041685.1029911>
- [54] C. Michael and S. Lavenhar, “Source Code Analysis Tools - Overview,” May 2013, accessed: 11.10.2017. [Online]. Available: <https://www.us-cert.gov/bsi/articles/tools/source-code-analysis/source-code-analysis-tools---overview>
- [55] “Coverity,” accessed: 26.9.2017. [Online]. Available: <http://www.coverity.com/>
- [56] “Coverity scan: Libreoffice,” accessed: 17.12.2017. [Online]. Available: <https://scan.coverity.com/projects/211>
- [57] “Coverity scan: Linux,” accessed: 17.12.2017. [Online]. Available: <https://scan.coverity.com/projects/linux>
- [58] A. Michalas and R. Murray, “Keep Pies Away from Kids: A Raspberry Pi Attacking Tool,” in *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, ser. IoTS&P ’17. New York, NY, USA: ACM, 2017, pp. 61–62. [Online]. Available: <http://doi.acm.org/10.1145/3139937.3139953>
- [59] K. Shaukat, A. Faisal, R. Masood, A. Usman, and U. Shaukat, “Security quality assurance through penetration testing,” in *2016 19th International Multi-Topic Conference (INMIC)*, December 2016, pp. 1–6.
- [60] M. Denis, C. Zena, and T. Hayajneh, “Penetration testing: Concepts, attack methods, and defense strategies,” in *2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, April 2016, pp. 1–6.
- [61] G. Lyon, “Nmap - Network Mapper,” August 2017, accessed: 10.8.2017. [Online]. Available: <https://nmap.org/>
- [62] J. Dawson and J. T. McDonald, “Improving Penetration Testing Methodologies for Security-Based Risk Assessment,” in *2016 Cybersecurity Symposium (CYBERSEC)*, April 2016, pp. 51–58.
- [63] M. R. Reddy and P. Yalla, “Mathematical analysis of Penetration Testing and vulnerability countermeasures,” in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, March 2016, pp. 26–30.
- [64] A. G. Michael Sutton and P. Amini, *Fuzzing Brute Force Vulnerability Discovery*, 1st ed. Addison-Wesley, 2007.
- [65] Google, “Continuous Fuzzing for Open Source Software,” 2017, accessed: 16.7.2017. [Online]. Available: <https://github.com/google/oss-fuzz>

- [66] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, 1st ed. Artech House, Information Security and Privacy Series, 2008.
- [67] Google, “libfuzzer Tutorial,” 2017, accessed: 24.9.2017. [Online]. Available: <https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md>
- [68] A. Kettunen, “Browser bug hunting - Memoirs of a last man standing.” 44CON, 2013, accessed: 16.7.2017. [Online]. Available: <https://vimeo.com/109380793>
- [69] W. Chunlei, L. Li, and L. Qiang, “Automatic fuzz testing of web service vulnerability,” in *2014 International Conference on Information and Communications Technologies (ICT 2014)*, May 2014, pp. 1–6.
- [70] S. Gorbunov and A. Rosenbloom, “Autofuzz: Automated network protocol fuzzing framework,” *IJCSNS*, vol. 10, no. 8, p. 239, 2010.
- [71] G. Grieco, M. Ceresa, and P. Buiras, “Quickfuzz: An automatic random fuzzer for common file formats,” in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016. New York, NY, USA: ACM, 2016, pp. 13–20. [Online]. Available: <http://doi.acm.org/10.1145/2976002.2976017>
- [72] OWASP, “Fuzzing,” accessed: 23.9.2017. [Online]. Available: <https://www.owasp.org/index.php/Fuzzing>
- [73] OWASP, “OWASP Testing Guide Appendix C: Fuzz Vectors,” accessed: 23.9.2017. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors
- [74] P. Pietikäinen, A. Helin, R. Puuperä, J. Luomala, A. Kettunen, and J. Röning, “Security Testing of Web Browsers,” January 2011, accessed: 5.8.2017. [Online]. Available: <http://www.cloudsw.org/issues/2011/1/1/communications-of-the-cloud-software/ec2266cf-8d22-4bfe-a70c-3fa1569c7007>
- [75] A. Helin, “Radamsa,” 2017, accessed: 29.7.2017. [Online]. Available: <https://github.com/aoh/radamsa>
- [76] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, *Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols*. Cham: Springer International Publishing, 2015, pp. 330–347. [Online]. Available: https://doi.org/10.1007/978-3-319-28865-9_18
- [77] P. Godefroid, M. Y. Levin, and D. Molnar, “Automated Whitebox Fuzz Testing,” February 2008, pp. 151–166.

- [78] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: Whitebox Fuzzing for Security Testing,” *Commun. ACM*, vol. 55, no. 3, pp. 40–44, March 2012. [Online]. Available: <http://doi.acm.org/10.1145/2093548.2093564>
- [79] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based Greybox Fuzzing As Markov Chain,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: ACM, 2016, pp. 1032–1043. [Online]. Available: <http://doi.acm.org/10.1145/2976749.2978428>
- [80] R. Ma, W. Ji, C. Hu, C. Shan, and P. Wu, “Fuzz testing data generation for network protocol using classification tree.” IET Conference Proceedings, May 2014.
- [81] H. P. Patrice Godeferoid and R. Singh, “Learn&Fuzz: Machine Learning for Input Fuzzing,” *CoRR*, vol. abs/1701.07232, 2017. [Online]. Available: <http://arxiv.org/abs/1701.07232>
- [82] M. Zalewski, “American Fuzzy Lop,” July 2017, accessed: 17.7.2017. [Online]. Available: <http://lcamtuf.coredump.cx/afl/>
- [83] M. Zalewski, “Binary fuzzing strategies: what works, what doesn’t,” August 2014, accessed: 17.7.2017. [Online]. Available: <https://lcamtuf.blogspot.fi/2014/08/binary-fuzzing-strategies-what-works.html>
- [84] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based Whitebox Fuzzing,” *SIGPLAN Not.*, vol. 43, no. 6, pp. 206–215, June 2008. [Online]. Available: <http://doi.acm.org/10.1145/1379022.1375607>
- [85] “Robot Framework,” accessed: 28.9.2017. [Online]. Available: <http://robotframework.org/>
- [86] J. Engblom, “The Schiaparelli Lesson - Unusual and Faulty Conditions,” January 2017, accessed: 18.7.2017. [Online]. Available: <https://software.intel.com/en-us/blogs/2017/01/11/the-schiaparelli-lesson-unusual-and-faulty-conditions>
- [87] “Jenkins,” accessed: 22.9.2017. [Online]. Available: <https://jenkins.io>
- [88] OUSPG, “Radamsa,” accessed: 29.7.2017. [Online]. Available: <https://www.ee.oulu.fi/roles/ouspg/Radamsa>