Security Testing in Continuous Integration Systems

Mariana Cristina Figueiredo Paulo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors:  Prof. Miguel Nuno Dias Alves Pupo Correia
             Eng. José Carlos de Jesus Ramos

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

October 2016
There is no royal road to anything. One thing at a time, all things in succession. That which grows fast, withers as rapidly. That which grows slowly, endures.

Josiah Gilbert Holland
Acknowledgments

Closing the academic cycle of my life, I have spent the last 6 years as a student at Instituto Superior Técnico. I must thank this faculty for giving me a well rounded basis to start a new chapter of my life, not only on a technical but also on a personal note. Through all the ups and down I faced here, I now know it can only be a good thing which helped me grow and evolve. I will remember the experiences I had here and take them as a valuable lesson during the course of my life.

During the last year, as I was doing my MSc thesis, Professor Miguel Pupo Correia was always ready to give me prompt feedback and discuss the choices I was making throughout every step of this work. Thank you for all the advice and time and for giving me the chance to work in this area, discovering what I really enjoy doing.

Also, I must express my sincere gratitude to Nokia - Portugal, namely to my co-supervisor Engineer José Ramos and the Nokia R&D team. To José, thank you for always taking some time to guide me through this process and to share your knowledge. To the team, thank you for welcoming and receiving me in an amazing way. I will always value the 9 months I spent with you and all the experiences we have shared. I really appreciate everything you have done to help me succeed.

Finally, I must also thank my family and friends. Since I do not want to write too much, I will simply say I really do appreciate all the moments you took to encourage me and to keep me smiling. You’ve provided me the best education and support system I could ask for. So, wherever I am and whatever I do, I will keep working on making you proud.
Resumo

O uso de aplicações web tem vindo a aumentar, de forma massiva, no decorrer das últimas décadas. Com isto, e dado o crescente número de recursos disponíveis, os ataques que surgem acompanham a tendência e vão-se tornando cada vez mais sofisticados. Perante esta realidade, as empresas são obrigadas a estar, constantemente, em busca de novas e melhores formas de fornecer produtos seguros aos seus clientes.

O presente trabalho foi desenvolvido na Nokia – Portugal tendo por objectivo planejar e desenvolver um conjunto de testes de segurança que permitisse facilitar a detecção de vulnerabilidades enquanto os produtos ainda se encontram em fase de desenvolvimento. Posteriormente, os testes desenvolvidos seriam incorporados num sistema de integração contínua onde são executados automaticamente. Finalmente, depois do seu completo desenvolvimento, automação e inclusão na integração contínua, estes testes formaram o SQM Security Pipe.

Palavras-chave: testes de segurança, varrimento de vulnerabilidades web, varrimento de vulnerabilidades, varrimento de portos, automação, integração contínua
Abstract

The usage of web applications has had a massive growth over the last decades. Together with the ever increasing number of resources available, the attacks performed keep up with this tendency and keep getting more and more sophisticated. Knowing this, companies strive to provide secure products to their clients and are constantly in search of new and better methodologies to do so.

The present work was developed at Nokia – Portugal aiming to design and develop a set of security tests that would facilitate the detection of vulnerabilities while products are still in the development phase. These tests would, then, be part of a continuous integration system where they are executed automatically. In the end, after being fully developed and automated and taking part in the continuous integration system, this work comes together as the SQM Security Pipe.

Keywords: security testing, web vulnerability scanning, vulnerability scanning, port scanning, automation, continuous integration
# Contents

Acknowledgments ................................................. v
Resumo ................................................................ vii
Abstract .......................................................... ix
List of Tables ........................................................ xiii
List of Figures ....................................................... xv
Glossary .............................................................. xvii

1 Introduction .......................................................... 1
   1.1 Motivation ..................................................... 1
   1.2 Objectives .................................................... 2
   1.3 Project Overview ........................................... 2
   1.4 Thesis Outline .............................................. 3

2 Related Work ...................................................... 5
   2.1 Black-box Security Testing ................................. 5
      2.1.1 Vulnerability Scanning ............................... 5
      2.1.2 Fuzzing/Attack Injection ............................. 8
   2.2 White and Grey-box Security Testing .................. 10
      2.2.1 Static Code Analysis ................................. 10
      2.2.2 Grey-box Security Testing ......................... 11
   2.3 Threat Modeling ............................................ 13
      2.3.1 Threat Modeling - Attack Trees and Attack Modeling . 14
      2.3.2 Threat Modeling-based Security Testing .......... 16
   2.4 Model-based Security Testing ............................ 18
      2.4.1 Model-based Testing ................................. 19
      2.4.2 Model-based Security Testing ..................... 22
   2.5 Open Security Test Methodologies ...................... 25
   2.6 Continuous Integration .................................... 27

3 SQM Security Pipe ............................................... 29
   3.1 Environment ................................................ 29
      3.1.1 Service Quality Manager ............................ 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2</td>
<td>Robot Testing Framework</td>
<td>30</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Merlin Product Pipe</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Threat Modeling SQM</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Data-Flow Diagrams</td>
<td>31</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Threat Identification</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Integrated Security Tools</td>
<td>35</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Arachni - Web Vulnerability Scanner</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Nessus - Vulnerability Scanner</td>
<td>36</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Nmap - Port Scanner</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Test Development</td>
<td>37</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Web Vulnerability Tests</td>
<td>37</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Vulnerability Tests</td>
<td>40</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Port Scanning Tests</td>
<td>41</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Final Work Summary</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>Evaluation</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Previous Solution - STAS</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>Developed Solution</td>
<td>48</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Qualitative Evaluation</td>
<td>49</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Quantitative Evaluation</td>
<td>50</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Stakeholder Evaluation</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Achievements</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>55</td>
</tr>
<tr>
<td>A</td>
<td>Test Example</td>
<td>59</td>
</tr>
<tr>
<td>B</td>
<td>Confluence Page Example</td>
<td>69</td>
</tr>
<tr>
<td>B.1</td>
<td>Main Confluence Page Example</td>
<td>69</td>
</tr>
<tr>
<td>B.2</td>
<td>Web Vulnerability Tests Confluence Page Example</td>
<td>71</td>
</tr>
</tbody>
</table>
## List of Tables

2.1 Tools Under Benchmarking adapted from [3] ............................................. 7
2.2 Tools studied in Sections 2.1 and 2.2 ....................................................... 13

3.1 Threat Identification of the SQM components .......................................... 35
3.2 Developed Test Suites and Test Cases Summary ....................................... 43
3.3 Developed Test Cases and Keywords Summary ......................................... 44

4.1 Security Cycle Metrics - before the STAS solution, with the STAS solution, with the SQM Security Pipe solution ......................................................... 50
List of Figures

2.1 Basic attack tree for opening a room electronically locked adapted from [22] .................. 15
2.2 MBT’s fundamental tasks  ................................................................................................. 19

3.1 Context Diagram for SQM  .............................................................................................. 31
3.2 Level 1 Data-flow Diagram for SQM ................................................................................ 32
3.3 Level 2 Data-flow Diagram for SQM ................................................................................ 34
3.4 SQM Production Pipe ...................................................................................................... 37
3.5 SQL Injection - test case example .................................................................................... 39
3.6 Arachni’s blacklist example ............................................................................................. 40
3.7 TCP Scan - test case example ......................................................................................... 42
# Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASVS</td>
<td>Application Security Verification Standard</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration</td>
</tr>
<tr>
<td>CSRF</td>
<td>Cross-Site Request Forgery</td>
</tr>
<tr>
<td>CVE</td>
<td>Common Vulnerabilities Exposures</td>
</tr>
<tr>
<td>DFD</td>
<td>Data-Flow Diagram</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>MBST</td>
<td>Model-based Security Testing</td>
</tr>
<tr>
<td>MBT</td>
<td>Model-based Testing</td>
</tr>
<tr>
<td>MPP</td>
<td>Merlin Product Pipe</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology of the U.S. Department of Commerce</td>
</tr>
<tr>
<td>NVD</td>
<td>National Vulnerability Database</td>
</tr>
<tr>
<td>RCA</td>
<td>Root Cause Analysis</td>
</tr>
<tr>
<td>SAMATE</td>
<td>Software Assurance Metrics and Tool Evaluation</td>
</tr>
<tr>
<td>SDL</td>
<td>Security Development Lifecycle</td>
</tr>
<tr>
<td>SQLi</td>
<td>SQL Injection</td>
</tr>
<tr>
<td>SQM</td>
<td>Service Quality Manager</td>
</tr>
<tr>
<td>STAS</td>
<td>Security Test Automation Tool</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>TEX</td>
<td>Test Execution Server</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WASC</td>
<td>Web Application Security Consortium</td>
</tr>
<tr>
<td>WASSEC</td>
<td>Web Application Security Scanner Evaluation Criteria</td>
</tr>
<tr>
<td>XCS</td>
<td>Cross Channel Scripting</td>
</tr>
<tr>
<td>XSS</td>
<td>Cross Site Scripting</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

More than thirty years ago, software testing was already considered an important part of a software's lifecycle, with approximately 50 percent of the elapsed time as well as 50 percent of the total project cost being dedicated to testing the program or system being developed [1].

Among the several existing software testing categories there is security testing. In a very simple and synthesized definition, the main goal of software security is to assure a proper software behavior if the system suffers a malicious attack [2]. Security testing aims to provide this throughout multiple stages of a product's lifecycle, applying some of its techniques depending on what needs to be tested.

Attacks are becoming more sophisticated and, consequently, it is important to keep up-to-date notions of which vulnerabilities matter the most at a certain time [2] in order to know what to look for, how to look for it and which tools to use to do so. White, grey and black-box testing, as well as threat modeling and model-based security testing are all attempts at improving software security. All of these security testing methodologies go through software – following different methods – and return a report stating the vulnerabilities that were found.

Obviously, there are advantages and disadvantages associated with each testing approach, much related to the type of vulnerabilities that are and are not detected and with the effort required. Further information into this matter will be developed in Chapter 2, where each of these methodologies will be analyzed in more detail.

In addition, it is important to be aware that the later the vulnerabilities are discovered, the more difficult and costly it will be to mitigate them. Knowing this, a product's security should be managed during the software development phase and this usually happens, but not in the most effective manner. Consequently, what tends to happen is, after the software development phase, whenever a vulnerability is made public, the development team has to release a patch to fix it. According to Antunes and Vieira, "to prevent vulnerabilities, developers should apply coding best practices, perform security reviews of the code, use static code analyzers, execute penetration tests, etc. However, most times, developers focus on the implementation of functionalities to satisfy the user's requirements and the time-to-market
constraints, thus disregarding security aspects” [3].

1.2 Objectives

Nokia wants to make its products stand out from the security point of view. With this in mind, it is now not only developing new security products for mobile operators but also enhancing the robustness of the current products. Consequently, this implies increasing and optimizing the security tests that are currently being performed.

The proposed work aimed to design and develop a set of security tests that would facilitate the detection of vulnerabilities while products are still in the development phase. These tests would, then, be part of a continuous integration system where they are executed automatically. In a more thorough description, the aims of this work were to:

- Elaborate a description of the tests which would detect flaws in the fields of confidentiality, integrity, availability, authorization and non-repudiation;
- Implement the tests using the Robots Testing Framework (described in Section 3.1.2) and Python;
- Integrate the tests in Nokia’s continuous integration environments using Merlin Production Pipe (MPP) (described in Section 3.1.3);
- Evaluate the set of tests that could be applied to any product against the tests that are product specific.

Considering the latter, we mainly focused on working with SQM (Service Quality Manager)\(^1\), a Nokia product which “identifies how service quality for key services, such as voice over LTE or video, is being degraded”. So, after being developed, automated and taking part in the continuous integration system, this work comes together as the SQM Security Pipe.

1.3 Project Overview

With the previously defined objectives in mind, the developed project was divided in 3 main stages:

1. Planning what to test and how to test it;
2. Developing the planned tests;
3. Evaluating the developed tests.

Regarding the first stage, a threat modeling approach was used in order to get a more in-depth knowledge of the vulnerabilities that could exist and their respective classification in terms of the confidentiality, integrity, availability, authorization and non-repudiation fields mentioned before. After this, we evolved\(^1\)\url{https://networks.nokia.com/solutions/service-quality-manager}
into understanding how the security testing tools available could fit into identifying these possible flaws, and from there we planned what we could test and how to do so.

Secondly, and making use of the plans made, we studied how the chosen security testing tools could be integrated into the robot framework. From there, and using this framework, the tests were developed, always aiming for the biggest automation level possible, since the goal was that the tester had little to no effort when running the tests. In the end, 3 test suites were developed: a web vulnerability test suite, a vulnerability test suite and a port scanning test suite. Each of these has several test cases which correspond to the different types of tests, under the scope of the respective test suite, that we wanted to perform.

Finally, once the tests were developed, we evaluated which were the advantages brought by this work. Not only against the previously existing work, but also against when security tests automation did not exist. Adding to this, we briefly evaluated how this work, which was developed with regard to SQM, could be applied to other existing products.

1.4 Thesis Outline

To get a better understanding of the proposed work, Chapter 2 presents some of the existing testing methodologies and, also, some continuous integration solutions. Section 2.1 will go further into black-box testing, focusing on vulnerability scanning and fuzzing/attack injection. Then, in Section 2.2, we will move into white and grey-box security testing with special attention to code analysis and grey-box testing itself. In Section 2.3, we will talk about threat-modeling based security, introducing the threat modeling concept first and, afterwards, evolving into the threat modeling-based security testing. In Section 2.4 we will step into model based security testing, following the same approach taken in Section 2.3, while in Section 2.5 we will focus on open security test methodologies. To end Chapter 2, we will introduce continuous integration in Section 2.6.

Chapter 3 aims to describe the developed work in further detail. Section 3.1 starts by briefly presenting the Nokia products that were the key components of this work. Then, Section 3.2 goes into the threat modeling performed on SQM being followed by Section 3.3 which introduces the security tools used to perform the tests and, finally, Section 3.4 explains the details of the developed tests.

Chapter 4 states how the developed work improved from the previously existing solution and, also, which are the advantages it brings to SQM's development. Section 4.1 describes STAS, the solution that was used previously and, in addition, reasons about why it was not used in the scope of this work. Then, Section 4.2 presents the evaluation of the developed SQM Security Pipe both in a qualitative (Section 4.2.1) and in a quantitative manner(Section 4.2.2).

Finally, Chapter 5 sums up how the initial goals were fulfilled in Section 5.1 and, in Section 5.2, points out what can still be improved.
Chapter 2

Related Work

Aiming to understand the field of software security testing this chapter presents some of the existing testing methodologies, along with examples of their employment. In addition, continuous integration is also introduced, being an important part of this work.

2.1 Black-box Security Testing

Black-box security testing consists in testing a program without being aware of its inner details and/or workings. In a more thorough manner, black-box testing takes a running program and analyzes it by sending a big amount of inputs with various formats [2], mainly focusing on the input formats and not on the software being tested [4].

The main advantage of this approach, as it was referred above, is that the tester does not need to know the specifics of a program. Due to this, it is a popularly used technique. Still, at the same time, this becomes its biggest disadvantage, since if the code itself is not considered, then a lot of vulnerabilities may not even be analyzed.

In the following sub-sections, some vulnerability scanning tools will be presented, as well as some fuzzing/attack injection tools, defining some methods and stating some of the existing resources that follow a black-box approach.

2.1.1 Vulnerability Scanning

Vulnerability scanning stands for using an automated tool to analyze web applications to verify the existence of vulnerabilities without access to the application’s source code [5]. It is important to highlight that vulnerability scanners perform attack injections relying on a vulnerability database that contains all the necessary information to perform the intended attack.

The Web Application Security Consortium (WASC)\(^1\) has produced a document that defines guidelines to evaluate vulnerability scanners on their efficiency testing web applications as well as identifying vulnerabilities [6], the WASSEC (Web Application Security Scanner Evaluation Criteria). Since web

\(^1\)www.webappsec.org
application security scanners have become increasingly important due to the proliferation of web application vulnerabilities, it is important to be able to understand which tools are more appropriate for each case. For this reason, NIST’s (National Institute of Standards and Technology of the U.S. Department of Commerce) SAMATE (Software Assurance Metrics and Tool Evaluation) project has also issued a document specifying the functional behavior of web application security scanners [7].

The WASSEC is a more general guide, providing a list of features that help with evaluating web application scanners so that a user can choose the scanner that best fits his needs. On its hand, the SAMATE publication explains how to measure the capability of a certain tool to meet the software assurance needs of a user by specifying basic functional requirements for web application security scanner tools.

The criteria, as defined in the WASSEC, that should be taken into consideration when evaluating a web application security scanner are:

- Protocol Support;
- Authentication;
- Session Management;
- Crawling;
- Parsing;
- Testing;
- Command and Control;
- Reporting.

Adding to this, the SAMATE publication defines task requirements for mandatory and optional features. In order to achieve a baseline capability all mandatory features must be met. So, in a high level description, “a web application security scanner shall be able to (at minimum): identify specified types of vulnerabilities in a web application; for each vulnerability identified, generate a text report indicating an attack; identify false positive results at an acceptable low rate” [7].

Despite of these requirements, the first and most important thing before evaluating any web application is to define which are its security requirements. Namely, the web application’s scenario and the types of vulnerabilities to detect since the criteria applied depends on that to classify a tool’s performance.

In addition to these well defined criteria, there are other methods for evaluating vulnerability scanners. Antunes and Vieira presented a method to do so, by assessing and comparing vulnerability scanners for web services in [3] with a benchmarking approach and examples. The referred approach particularly focuses on two metrics: precision (ratio of correctly detected vulnerabilities over the number of all detected vulnerabilities) and recall (ratio of correctly identified vulnerabilities over the number of all known vulnerabilities), using them to define benchmarks that evaluate and compare alternative tools [3]. Two benchmarks were defined:
<table>
<thead>
<tr>
<th>Tool Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability Scanner</td>
<td>HP - WebInspect</td>
</tr>
<tr>
<td></td>
<td>IBM - Rational AppScan</td>
</tr>
<tr>
<td></td>
<td>Acunetix - Web Vuln. Scanner</td>
</tr>
<tr>
<td></td>
<td>Univ. Coimbra - IPT-WS</td>
</tr>
<tr>
<td>Static Code Analyzer</td>
<td>Univ. Maryland - FindBugs</td>
</tr>
<tr>
<td></td>
<td>SourceForge - Yasca</td>
</tr>
<tr>
<td></td>
<td>JetBrains - IntelliJ IDEA</td>
</tr>
<tr>
<td>Runtime Anomaly Detector</td>
<td>Univ. Coimbra - CIVS-WS</td>
</tr>
</tbody>
</table>

Table 2.1: Tools Under Benchmarking adapted from [3]

- **VDBenchWS-pd** - based on a predefined workload, regards tools able to detect SQL Injection vulnerabilities in SOAP web services using techniques like penetration testing, static code analysis and runtime anomaly detection;

- **PTBenchWS-ud** - based on a user-defined workload, specifically regards penetration testing tools able to detect SQL Injection vulnerabilities in SOAP web services.

The main difference between the two benchmarks is that the first is based on a well-defined set of rules whilst the second allows the benchmark user to specify the workload, meaning that it is not predefined beforehand and is unknown to the tools’ providers.

Under the defined benchmarks, 4 vulnerability scanners, 3 static code analyzers and 1 runtime anomaly detector – listed in Table 2.1 – were put to test and were ranked under the criteria of fulfilling the properties of repeatability, portability, representativeness, non-intrusiveness and simplicity of use [3].

The results confirmed that the benchmarks presented may be used to assess and compare penetration testers, static code analyzers and anomaly detectors in a successful manner.

In the work presented by Bau et al. in [5], the goal was to assess the current state of the art of automated black-box web application vulnerability testing. The referred assessment consisted in answering three questions:

1. Which vulnerabilities are tested by the scanners?
2. How representative are the scanner tests of vulnerability populations in the wild?
3. How effective are the scanners?

By testing eight well-known commercial vulnerability scanners against a common set of sample applications the findings were that:

- The vulnerabilities mostly targeted by the scanners are, in order, Information Disclosure, Cross Site Scripting (XSS), SQL Injection and other forms of Cross Channel Scripting (XCS);
• Many scanners are effective at following links whose targets are textually present in served pages but, in contrast, most scanners are not effective at following links through active content technologies such as Java applets, SilverLight and Flash;

• In general, the scanners are effective at detecting well-known vulnerabilities;

• Regarding the detection of stored vulnerabilities – i.e. stored XSS/SQLi vulnerabilities where unsanitized user input is written to the database and which executes when loaded – the scanners are very inefficient.

So the results show that these scanners mainly devote their attention to information leakage vulnerabilities, followed by XSS and SQLi vulnerabilities.

2.1.2 Fuzzing/Attack Injection

Fuzzing is a brute force technique that stands for detecting vulnerabilities by feeding software with unexpected input and checking for resulting exceptions [8]. The concept consists in having a program generate inputs that are run against the system being tested. In a six steps description, adapted from [8]: first and before anything else, the target must be identified in order to select a fuzzing tool or technique; secondly, all inputs must be identified, since missing an input entry point might cause missing a vulnerability in the program; after this identification phase, the fuzzed data has to be generated – so this will be step 3 – and executed, which will be step 4; in fifth place, and not always being a considered step, the fuzzer must monitor the program for exceptions or faults, so that the input that caused it can be identified; finally, as the sixth and final step, once the fault/exception cause is identified, it is necessary to determine if the bug can be further exploited.

Although fuzzing and vulnerability scanning are often confused and are, in fact, similar techniques, there are subtle differences. While fuzzing discovers bugs by sending a large amount of erroneous inputs to the system and monitoring if it fails, vulnerability scanning tries different attacks that are known to, a priori, be able to exploit some specific vulnerabilities. In addition, vulnerability scanners are usually commercial tools, whereas fuzzers are frequently open source. Still, the fact that fuzzing uses a brute force approach while vulnerability scanning follows an established method, as explained in Section 2.1.1, remains their main difference, as referred in [9].

Fuzzing can, for instance, identify access control vulnerabilities which represent severe security threats, since attackers can exploit them and get information stored in databases that should be protected. Li et al. presented BATMAN [10], a prototype using a cross-platform black-box technique for identifying this kind of vulnerability. Its functioning is as follows:

1. The access control policy is modeled through a virtual SQL query which is able to capture the database accesses and the post-processing filters within the application;

2. After the modeling, a crawler explores the application and collects execution traces. From those it identifies the allowed database accesses for each role and – over the operation parameters – extracts the constraints to characterize the relationship between the users and the accessed data.
Based on this inferred policy, two types of test inputs are generated to exploit the application for potential access control flaws:

- **Role-based tests** - which check if a user with less privileges can trigger a database access operation that requires a more privileged role;

- **User-based tests** - which check if a user can trigger a database access operation with a parameter that is subject to the user-based constraint but that takes a value that is linked to another user.

Despite of some limitations related to the inference technique and the web response analysis, the results showed that this approach can bring very accurate and effective results.

Besides fuzzing, attack injection is also a technique that stands for detecting vulnerabilities in software. Still, with attack injection the generated input is not random, since it is not a brute force approach. AJECT (Attack inJECtion Tool) [11] was implemented using an attack injection methodology with the goal of automatically discovering software vulnerabilities. The chosen approach was to access an interface and stress it with interactions which do not follow the usually expected behavior. Unlike the previously presented method, here the attacks are created by making use of a specification of the server’s communication protocol and predefined test case generation algorithms. After this, the tool injects the attacks and monitors the evolution of the component or system in order to detect unexpected behaviors. As soon as one of these is observed, indicating that a vulnerability might exist, evidence about the attack is gathered and can be used to remove the referred vulnerability. AJECT was tested against up-to-date IMAP and POP3 e-mail servers, fully patched to the latest stable version and with no known vulnerabilities. In spite of that, AJECT was able to find new vulnerabilities in 5 of the 16 tested e-mail servers, being that these were detected either by a crash or a denial-of-service.

A problem with following a black-box approach is that when the application under test is very complex and has multiple actions that can change its state, the state change is not considered and vulnerabilities can be missed in that portion of the web application [12]. The work proposed by Doupé et al. has a new way of inferring the web application’s internal state and still follow a black-box approach. It navigates through the web application while observing differences in the output and incrementally producing a model that represents the web application’s state. The stage-change is detected by providing the same input and getting a different output.

When the web application's state machine is inferred, that information along with the list of request-responses made by the crawler is used to perform a state-aware fuzzing of the web application, looking for security vulnerabilities. Comparing to other approaches already in use, the results showed that this approach had the best code coverage for every application, thus validating the idea that understanding the state is necessary to better exercise a web application.

The biggest advantage of fuzzing is that it augments the efforts of a person manually testing a software since any fuzzer, even the most basic kind, can outnumber the amount of input sets used in a manual validation process [13]. The problem is that when fuzzers operate using purely random data their efficiency is compromised and this should be regarded when using this security testing technique.
2.2 White and Grey-box Security Testing

As it can be inferred from Section 2.1, apart from black-box security testing being a widely used methodology, there are several types of vulnerabilities that it does not identify, since they are not even considered. Grey and white-box security testing solve this problem by concentrating on the vulnerabilities that the black-box approach does not consider.

This is possible because when using a white-box technique (Section 2.2.1), as opposed to a black-box one, the program's code is accessible, so the coding vulnerabilities can be addressed. Grey-box security testing, as its name suggests, follows a method that has both black and white-box security testing characteristics and it will be further discussed in Section 2.2.2.

2.2.1 Static Code Analysis

Static Code Analysis is a white-box approach, since it leverages the source, binary or intermediate code to identify possible vulnerabilities. This identification is based on already known risks - typically put together in a vulnerability database - associated with the use of certain coding practices [14]. The basis of most code analysis tools is to read the source code and splitting it into tokens while searching for functions that are already recognized as dangerous practices that can be exploited [15]. As the tool finds usages of the dangerous functions, it shows an error message stating the problem.

ITS4, developed by Cigital\(^2\) in the mid 1990's, was – according to the company’s website – the first ever commercial static analysis tool. It followed the idea of preprocessing and tokenizing source files and then matching the resulting tokens against a library of vulnerabilities. It now serves as a basis for some better and much evolved approaches. Chess and McGraw discuss that “to increase precision, a static analysis tool must leverage more compiler technology. By building an abstract syntax tree (AST) from source code, such a tool can take into account the basic semantics of the program being evaluated” [15]. After obtaining the AST and to keep increasing the tool’s precision, it is important to decide if the performed analysis will be of local, module-level or global scope. The differences between the three scopes, respectively and as is defined in [15], are if the program is examined: one function at a time and does not consider relationships between functions; a class or compilation unit at a time and so relationships between functions in the same module as well as properties that apply to classes are considered but not calls between modules; the whole program is taken into account, so all relationships between functions are considered. The broader the scope the more context amount is considered by the tool, so this has to be taken into consideration to keep a balance between reducing false positives (with a bigger context amount) and having a huge amount of computation to do.

Static analysis has many ways of being applied. Wasserman and Su presented a static analysis tool for finding cross-site scripting (XSS) vulnerabilities addressing the problem of weak or even absent input validation [16]. The chosen approach is divided into two parts:

- a string analysis able to track untrusted substrings, *i.e.* a string-taint analysis which will make use

\(^2\)www.cigital.com
of context free grammars to represent sets of string values and, also, model the semantics of string operations;

- untrusted scripts check based on formal language techniques - enforcing the policy that generated web pages include no untrusted scripts.

This proved to be a good method, having detected both known and unknown XSS vulnerabilities in real-world web applications.

Another work, by Jovanovic et al., presents Pixy\(^3\) with an integrated precise alias analysis for static detection of web application vulnerabilities which targets the unique reference semantics commonly found in scripting languages [17]. The argument used is that for static detection of vulnerabilities, the only way in which the results can be precise is by considering possible alias relationships between PHP variables. Two or more PHP variables are aliases if at a certain program point their values are stored at the same memory location. Considering this helps taint analysis to be able to decide if a certain assignment affects not only the assigned variable but also its aliased variable, turning them both into tainted variables.

Despite all the efforts, most of these static analysis tools are still known to report many false positives. In order to improve this, \textit{i.e.} to reduce the number of false positives, Medeiros et al. used data mining and presented an approach capable of automatically detecting and correcting web application vulnerabilities [14]. The novelty in the chosen approach is that adding to the analyses of the source code searching for vulnerabilities, as in the previously presented works, this tool also inserts fixes in the same code to correct the detected flaws. While this happens, the programmer is allowed to understand where the vulnerabilities were found and how they were corrected. This work's methodology was implemented as a sequence of four steps: \textbf{taint analysis} - which, ultimately, will generate trees describing candidate vulnerable control-flow paths; \textbf{data mining} - obtains attributes from the candidate vulnerable control-flow paths and uses a classifier to predict if each candidate vulnerability is a false positive or not; \textbf{code correction} - with the control-flow path trees of vulnerabilities predicted not to be false positives, identifies the vulnerabilities and the fixes to insert as well as the places where they have to be inserted. After this, it assesses the probabilities of the vulnerabilities being false positives and modifies the source code with the fixes; \textbf{feedback} - provides feedback to the programmer based on all of the collected data in the previous steps.

Static analysis cannot solve all the security problems, but combined with other methodologies it has proven to be an effective way of preventing vulnerabilities that are mostly based on limited programming skills and lack of security awareness on part of the developers [17].

\subsection*{2.2.2 Grey-box Security Testing}

As stated in the beginning of Section 2.2, grey-box security testing is a combination of black-box and white-box security testing methodologies. In this approach the tester uses his knowledge about the

\footnote{\url{https://github.com/oliverklee/pixy}}
internal structure of the SUT (system under test) to design the test cases (as in the white-box approach) but tests the application from a black-box perspective by running it and analyzing the results.

*Check 'n' Crash* (CnC) is a grey-box security tester for Java based programs which combines static checking and concrete test-case generation [18]. It is a combination of two tools that result in the referred combination of approaches used by a grey-box tester. ESC/Java, one of the two tools, is the static checking system that analyzes a program, reasons abstractly about unknown variables and tries to detect errors. Taking this, error reports are generated with reference to the abstract conditions that are necessary for that type of error to take place. Having the error reports, the second tool – JCrasher – produces Java test cases that will in fact activate the existing errors.

The principle applied consists in taking the names of the Java files to be tested, compiling them and formulating a logic predicate set which will specify the states from which the execution of each method will execute normally or not. When an execution goes wrong, CnC computes the variable assignments that caused the error and uses JCrasher to compile them into JUnit test cases. Finally, the test cases are executed under JUnit and if its execution does not produce any errors then it is identified as a false positive. If, on the other hand, the execution produces one of the previously identified errors, CnC generates an error report.

KLEE is another example of a tool capable of automatically generating tests. It uses symbolic execution to analyze applications, creating a high coverage set of test inputs [19]. By running programs symbolically, instead of getting concrete values produced by the program in a normal execution, it is possible to get constraints that describe in an exact manner the set of values possible on a given path. “When KLEE detects an error or when a patch reaches an exit call, KLEE solves the currents path’s constraints (...) to produce a test case that will follow the same path when rerun on an unmodified version of the checked program”.

As for SAGE (Scalable Automated Guided Execution), it adapted fuzzing – a traditionally black-box technique – to use symbolic execution and dynamic test generation, turning it into a grey-box testing tool or, as the authors preferred to name it, an automated white-box fuzz testing tool [20]. As it was covered in Section 2.1, even though fuzzing is a really effective technique, due to its black-box approach it has some limitations. Knowing this, SAGE uses a newly designed – by 2012, when the paper was published – search algorithm to eliminate those limitations called *generational search*. The main search procedure follows the standard, running the program with an initially fixed input (in a workList) to detect any bugs. Following this first run, the inputs are processed by selecting an element and expanding it to generate new inputs. For each of those childInputs, the program under test is run with that input. This execution is checked for errors and is assigned a score before being added to the workList which is sorted by these scores. In order to improve the search space size, this search algorithm starts with an initial input and an initial path constraint and attempts to expand all program constraints present in the initial path constraint. By using a bound to limit backtracking of the children sub-searches, the algorithm is able to reduce the search space.

This approach gathers input constraints by recording a run of the program with a well-formed input. The collected constraints are then negated one by one and solved with a constraint solver, producing
<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Approach</th>
<th>Technique</th>
<th>Language</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BATMAN</td>
<td>Black-box</td>
<td>Fuzzing</td>
<td>Java</td>
<td>[10]</td>
</tr>
<tr>
<td>Not specified (Extended Minamid's PHP string analyzer)</td>
<td>White-box</td>
<td>Static Analysis</td>
<td>O'Calm</td>
<td>[16]</td>
</tr>
<tr>
<td>Pixy</td>
<td>White-box</td>
<td>Static Analysis</td>
<td>Java</td>
<td>[17]</td>
</tr>
<tr>
<td>Check 'n' Crash</td>
<td>Grey-box</td>
<td>Static Analysis and Automatic Test Generation</td>
<td>Java</td>
<td>[18]</td>
</tr>
<tr>
<td>KLEE</td>
<td>Grey-box</td>
<td>Symbolic Execution and Automatic Test Generation</td>
<td>C++</td>
<td>[19]</td>
</tr>
<tr>
<td>SAGE</td>
<td>Grey-box</td>
<td>Symbolic Execution and Fuzzing</td>
<td>-</td>
<td>[20]</td>
</tr>
<tr>
<td>SecTAC</td>
<td>Grey-box</td>
<td>Trace-based Symbolic Execution and Satisfiability Analysis</td>
<td>-</td>
<td>[21]</td>
</tr>
</tbody>
</table>

Table 2.2: Tools studied in Sections 2.1 and 2.2

new inputs that will allow for the generation of new test cases in a fast and effective way.

Also based on leveraging a program’s symbolic execution, obtaining program constraints and security constrains, SecTAC was developed as a tool to detect vulnerabilities in C programs using trace-based testing [21]. “The novelty of this method is a test model that unifies program constraints and security constraints such that formal reasoning can be applied to detect vulnerabilities”. With this approach, using trace-based symbolic execution and satisfiability analysis, even if the existing tests do not trigger any vulnerabilities, the security flaw is still detected.

The advantage of using trace-based symbolic execution over conventional symbolic execution is that it avoids having a massive search space. First, it is applied to produce program constraints specifying a condition that program variables must satisfy after the execution of the statement. Second, it is applied to produce security constraints specifying the condition that program variables must satisfy to ensure the security of the given program execution [21]. With this data, formal reasoning can be performed to detect security vulnerabilities when an assignment of values to program variables satisfies the program constraints but does not satisfy the security constraints.

2.3 Threat Modeling

Threat Modeling stands for analyzing and describing the attacks that a system can suffer, considering its security-relevant features and its attack surfaces. More thoroughly, threat modeling intends to help with understanding: what the attack goals are, who the attackers are, what attacks are likely to occur, the security assumptions of a system and where to best spend a security budget [22]. As in other testing methodologies, the earlier the threat modeling process is started the better the problems will be found and fixed and, also, the less expensive it will be to deliver secure products.

There are several methodologies used to do this, some of the mostly used ones will be discussed in the following section.
2.3.1 Threat Modeling - Attack Trees and Attack Modeling

According to Microsoft, the SDL (Security Development Lifecycle) threat modeling process can be synthesized into four main steps: diagramming; threat identification; mitigation and validation [23] [24]. It aims to improve design security, to document the security design activity and to teach developers about security [24].

There has to be a process planning and relevant information gathering about the application that is relevant for the modeling. After this, and regarding the diagramming phase, the application can be decomposed into components that are both relevant on the architectural scope and on the possibility of being attack targets. The components decomposition should be done using DFDs (Data Flow Diagrams) because they include processes, data stores, data flows, trust boundaries and external entities and, so, they are considered to be data-centric and easy to understand. This helps with describing the boundaries of the system and can provide a good level of detail of the system components. Considering that a lot of software attacks involve the flow of data through the system, DFDs focus on the right aspect: information flow. In addition, the decomposition must be done in a hierarchical way, starting from level 0 which decomposes the application in a high level thus showing its main components and user types. After this, the decomposition can go onto the required level of detail but it is not recommended that it goes above level 3, since from that point on the level of detail and the number of aspects to be considered will be too complex, turning the process too slow.

The second step focuses on identifying threats for each of the diagram components, making use of a vulnerabilities taxonomy. STRIDE is the taxonomy used, containing 6 categories of vulnerabilities, each of them corresponding to a character from the taxonomy name.

- **Spoofing** - the authentication security property can be violated, granting the attacker the opportunity to act under a valid entity in the system which does not belong to him;

- **Tampering** - the integrity security property can be violated, enabling the attacker to, in an unauthorized way, modify data;

- **Repudiation** - the nonrepudiation security property can be violated, allowing the attacker to be able to deny having performed an action;

- **Information Disclosure** - the confidentiality security property can be violated, giving the attacker unauthorized access to data;

- **Denial of Service** - the availability security property can be violated, so the attacker can degrade and even deny a certain component's service;

- **Elevation of Privilege** - the authorization security property can be violated, giving the attacker capabilities to which he does not have authorization.

The third step is the main goal of doing threat modeling: mitigation. For each identified threat, there are several ways to address it, either by redesigning to eliminate it, applying standard mitigations (for example, Access Control Lists), applying new mitigations or, ultimately, accepting the vulnerability in
Open room electronically locked

Take door down  Pick e-lock  Learn combination

Access e-lock data  Physically access cables  Find written combination  Get combination from target

Threaten target  Bribe target

Figure 2.1: Basic attack tree for opening a room electronically locked adapted from [22] design. Following this, the fourth and final step validates the whole threat model, namely the graph analysis of diagrams, checking that the final diagrams reflect the final code, that STRIDE threats per element have been enumerated, that the whole threat model has been reviewed and that each threat is mitigated [24].

In the first SDL threat modeling approaches it was not present, but after realizing that a risk assessment technique needs to be applied, the DREAD technique was added. This technique provides a way of rating vulnerabilities by the order of risk that they represent. The risk factor is calculated through

\[
\text{risk} = \text{threat level} \times \text{vulnerability degree} \times \text{impact}
\]

DREAD, like STRIDE, is a mnemonic and each of its name characters represents a risk factor that can vary between 1 and 10 (being 10 the worst scenario):

- **Damage potential** - how harmful can it be if the vulnerability is successfully exploited;
- **Reproducibility** - how easy it is to exploit the vulnerability;
- **Exploitability** - level of skills and effort needed to exploit the vulnerability;
- **Affected users** - how many users may be affected by a successful exploitation of the vulnerability;
- **Discoverability** - how easy it is to discover the vulnerability.

The level of risk is, then, obtained by calculating the average value of the values assigned to each DREAD parameter. Each of these values is assigned based on a detailed description of the potential attacks against the vulnerability which comes from the corresponding attack tree [9]. The attack tree is, then, a way of thinking and describing security of systems and subsystems [22]. It represents the attacks to a given vulnerability and its countermeasures as a tree structure. In this structure, the root node represents the goal of the attack while leaf nodes represent the attacks (Figure 2.1 shows a simple example).

Leaf nodes can have various types which may be combined. “And” nodes represent different steps in achieving an objective (attack) whereas “Or” nodes represent different ways towards achieving the
same objective. Boolean nodes (for example, possible and impossible nodes, easy and not easy nodes, etc) are the ones that can be combined with the “And” and “Or” node types, resulting in a clear stating about the attacks. There are other types of nodes, like Continuous node values, which can translate the cost of attacking or defending a target, the time to achieve or repulse an attack or even the probability of success of a given attack.

As it was mentioned previously, each goal is a separate attack tree, being these goals identification the first step towards constructing a tree. The second step is, for each goal, identifying attacks against them and repeat this step as needed. After this, existing attack trees can be plugged in as appropriate.

Certain attack paths in the attack tree can be explored in more depth than others enabling the development of some attack analysis tools like the Topological Vulnerability Analysis (TVA) tool [25]. “To understand overall vulnerability to network attack, one must consider attacker exploits not just in isolation, but also in combination. That is, one must analyze how low-level vulnerabilities can be combined to achieve high-level attack goals”. Given the combinations of modeled exploits on a network, TVA discovers attack paths leading to specific network targets, allowing for an assessment of the true vulnerability of critical network resources.

Firstly, TVA goes through a network discovery phase gathering information and correlating it with an exploit rule base that was collected previously. The analysis phase is the following, submitting the resulting network attack model to be analyzed by a custom engine which models network attack behavior, based on exploit rules, and building a graph of condition dependencies between exploits. The return is a set of attack paths that goes from the initial network state to a pre-determined attack goal.

Analyzing how these low-level vulnerabilities can be combined by attackers to meet attack goals is a difficult task. Consequently, many other types of analysis tools fail at understanding this and do not identify severe vulnerabilities that come from this combined approach. Thus the importance of attack modeling to try and solve this problem by making clear how these combinations work and how serious they can be.

2.3.2 Threat Modeling-based Security Testing

Despite of several existing software techniques, most of them focus on identifying vulnerabilities in products that are already implemented. With threat modeling it is possible to identify vulnerabilities in an earlier development stage, making it easier and less expensive to fix them.

Following Microsoft’s threat modeling approach, described in the above section, Marback et al. presented a system intending to automatically generate security tests from threat trees [26]. To do so it analyzes threat trees created by the Microsoft Threat Modeling Tool⁴, placing the data into a tree data structure. Using a depth-first approach, this tree data structure is walked in order to produce test sequences, namely every events sequence that is capable of exploiting the vulnerabilities of a system. The generated test sequences are not executable, so the next step focuses on transforming the test sequences into executable tests, allowing test case developers to only consider applicable and necessary events in the sequence to bind to the executable test code and creating a list of parameters (associated

with actions) by iterating through the events in a sequence. For every sequence, an executable test script is produced and an input specification for each test is also provided. To reduce the number of inputs, they are divided into 4 groups: the valid inputs, which meet all the design requirements and let the program behave as expected; the handled invalid inputs, that are invalid and which should be handled by a program’s error handling type; the unhandled invalid inputs, which are not addressed since they do not represent a security risk although they expose a software flaw; the unhandled invalid inputs, which are addressed since they do represent a security risk, being the main focus of the input generation part of this paper. With these groupings, the test generator can choose which are and which are not relevant inputs, thus justifying this division.

The experimental results of using the tool to test a real web application generated a good amount of test sequences and an even greater amount of test cases with test inputs that were able to demonstrate that this approach is in fact effective towards discovering unmitigated threats. Adding to this, the fact that it did not present many false positives or negatives also showed that it can also be an effective security testing and verification technique.

Apart from attack trees, there are other approaches in use which have proved to be effective with formal modeling and verification of attack behaviors. Predicate/Transition (PrT) nets are one option, consisting of high-level Petri nets that are a mathematically-based methodology for modeling and verifying distributed systems [27]. The advantage of these is that they capture control and data flows as well as complex attacks with partially ordered actions. Based on this, Xu et al. has produced a work that aims to use PrT net-based threat models to perform automated security testing, allowing threat models to be reused for validating the implementation obtained [27]. Mainly, it focuses on using Threat Model-Implementation Description (TMID) specifications to generate executable security test code in an automated manner. Before going in more depth into the chosen methodologies, it is important to define that a TMID specification is a junction of a threat model – for example, the referred PrT net – and a Model-Implementation Mapping (MIM) description. On the one hand the threat model describes the existing possibilities for an attacker to violate a security goal while on the other hand the MIM description binds each individual element of the threat model to its implementation.

With a given TMID specification, that work will produce all the attack paths from the threat model, built according to the STRIDE classification system. Then, and according to the MIM description, it will convert these attack paths into executable code, enabling the generated security tests to be executed automatically.

In a more thorough description, in a threat net each attack path, i.e. security test, is made of an initial test setting \((M_0)\) and several test inputs \((t_1 \theta_1, ..., t_\theta n-1)\) which have the same amount of following expected states \((M_1, ..., M_{n-1})\) defined as test oracles. Attack transitions \(t_n\) and its resulting states \(M_n\) are not physical operations, but they are important since they represent the condition and state of the security attack/risk [27]. If a test oracle is false then the security test fails, meaning that the system is not threatened by the attack. If, on the other hand, the test oracle is true, then the security test has a successful execution and it means the system is in fact threatened by the attack. In addition, when the MIM description for the threat net is provided, each attack path is converted into executable code,
resulting in a test suite. The results from the two case studies have shown that security testing with formal modeling is very effective, having demonstrated a 90 percent vulnerability detection rate. Despite of this, it is also shown that it is very difficult to uncover vulnerabilities that are not considered in the threat models.

Another existing option, regarding threat modeling security testing, consists in detecting undesired threat behavior at runtime. Instead of opting for describing threats with attack trees or threat nets, Wang et al. work uses UML to model security threats during the design phase of a product. Threat behaviors are modeled using a UML sequence diagram and, then, these models drive the security testing of the final system.

By reviewing the key use cases of a product, it is possible to identify a threat scenario, since it is defined as an undesirable behavior of the system which can compromise the security goals [28]. Once sequence diagrams describe interactions between objects by focusing on the message sequences exchanged as well as their corresponding occurrence specifications, by having a use case scenario it is possible to model the behavior of the system. Towards threat scenario modeling, the initial phase consists in determining a security policy – that describes what is and what is not allowed – and, also, the potential threats to the system. Threat identification is treated as a part of the requirements analysis, having threats being modeled with UML sequence diagrams describing which interactions have to happen before an attacker can compromise the system. So, consequently, threat scenarios are represented by sequences of message exchanges and threat behavior is characterized by being a scenario of misuse case and also a sequence of object interactions. In a later phase of product development, the code can be tested against the defined threat model during the design phase so to ensure there are no vulnerabilities that were reintroduced.

The key security testing goal with this approach is to verify the existence of consistency between the threat specification and the code implementation. The execution of the program is treated as an event sequence of method calls and method executions. If any of the observed event sequences resemble a possible event sequence regarded in the threat model, then the threat still remains and an error message is produced to report the failure [28].

In spite of not having been tested with many real projects, it is believed that a big advantage of this approach is that it can be easily applied without security expertise, because as the knowledge of security threats is acquired the threats are ready to be modeled and used to check if the implemented code still exercises the previously modeled threats.

2.4 Model-based Security Testing

Testers usually have an idea of how the software is supposed to behave when facing certain conditions. The problem is that this knowledge used to remain in their heads and was not written in paper so that other people could leverage it too [29]. When models of a program's behavior started to become popular, model-based testing (MBT) came into the spotlight and presented a methodology that helps people who know what the program is supposed to do to state that knowledge. At the same time,
it helps people who do not know the software to understand how it is supposed to perform. Consequently, model-based security testing aims to develop this into the security perspective, defining security requirements and checking what is and what is not in compliance with them.

2.4.1 Model-based Testing

A software model describes the software’s intended behavior in various scenarios like input sequences acceptance, actions performed and conditions that the system is under. The main goal of having a model written down is that it becomes a shareable, reusable and precise description of the SUT. There are many forms of doing this depending on what aspects of the software’s behavior the tester wishes to describe [29]. Some of the most used software models are:

- **Finite State Machines** - applied when the program’s state defines which inputs are available at a certain time. Usually used only if the program’s model can be described within a small number of specific states;

- **Statecharts** - applied when modeling complex or real-time systems, being able to specify state-machines in a hierarchy;

- **Unified Modeling Language (UML)** - applied when describing very complicated behavior;

- **Markov Chains** - applied as state machines, but aim not only to generate tests but also to gather and analyze failure data in order to estimate, for example, reliability and mean time to failure measures;

- **Grammars** - applied when modeling specific types of software, like parsers, because it presents an easier and more compact representation.

The MBT approach has seven fundamental tasks synthesized in Figure 2.2. The first of these is understanding the system under test, which demands for a deep knowledge of what the software needs to perform. This means that the tester has not only to know the software itself but also the environment in which the software is in, accomplishing this by applying exploratory techniques and reviewing available documents. As referenced by the Encyclopedia of Software Engineering [29], the following activities can be performed:

- Determine the components/features that need to be tested based on test objectives;

- Start exploring target areas in the system;
• Gather relevant, useful documentation;
• Establish communication with requirements, design and development teams if possible;
• Identify the users of the system;
• Enumerate the inputs and outputs of each user;
• Study the domains of each input;
• Document input applicability information;
• Document conditions under which responses occur;
• Study the sequences of inputs that need to be modeled;
• Understand the structure and semantics of external data stores;
• Understand internal data interactions and computation;
• Maintain one living document: the model.

With this first task, model-based testers gather information that enables building adequate models. Following this, the second task consists in choosing the most suitable model, since there is no software model that is able to comprehend all software intents and purposes. It is important to mention that this encyclopedia states that there is little or no data published that can highlight a model that is far better than others. Despite of this, in order to better choose a model, there are some factors that need to be taken into consideration such as: the application domain; specific characteristics of the SUT, for example if the tester wants to discover if a system can deal with long, complicated and non-repetitive inputs’ sequences and/or if the goal of the tester is to evaluate if the software behaves as expected for all input combinations. All in all, it is the amount of knowledge about the model along with the knowledge of the application domain that determine which model is chosen [29].

The third task is about building the model. This is characterized by having testers define high-level state abstractions and then refine them into a state space. These state abstractions are defined by the inputs and information about the applicability of each input as well as the behavior caused by the input. Usually, the adopted methodology consists in making a list of inputs and, for each input, it should be documented the situations in which the input can and cannot be applied by the users – these are called input applicability constraints – as well as the situations in which, depending on the context of the input application, the input causes different behaviors (the input behavior constraints). These constraints represent scenarios that are in need of testing.

The fourth task stands for generating the tests. Depending on the nature of the model, i.e. its properties, it can be more or less difficult to do so. For example, for some models the only thing that is necessary is to have a simple knowledge of combinatorics in order to go through the combinations of conditions described in the model. Having generated the tests, the fifth task is about running the them. Tests can be run as soon as they are generated, but the typical approach is to run the tests after having a complete suite generated that meets a certain amount of adequacy criteria.
After running the tests it is fundamental to collect the results, being this the sixth task of the MBT. With this, it is possible to evaluate the test results and determine if, given the sequence of test inputs applied, the software returned the correct output or not. The seventh and final task consists in making use of the test results. With model testing the results not only show which tests have been run but also which ones have not been run, providing a way of determining the completeness of a test and the stopping criteria related to the model.

The three key concerns stated about this testing technique are:

1. **State space explosion** - mentioned in almost every model-based testing related paper. The first way to deal with it is through abstraction, putting complex data into a simple structure. Although this causes the loss of some information, when used in a wise manner, it can keep the important information and ignore the irrelevant part. The other option is exclusion, choosing to drop information without going through the effort of having to abstract it;

2. **Meeting coverage criteria** - there are not many (if any) studies about the efficiency or effectiveness of the various model coverage proposals;

3. **Oracles and automation** - oracles are crucial towards making test automation work, since they determine if a test behaves as it should or not. The problem is oracles need to be written manually and the more speed of automation is required the more functionality has to be covered during testing. This causes the oracles to become harder and harder to write, being that there is, as in the previous point, little investigation put into this problem.

Regarding the lack of studies to measure the efficiency of MBT, Dalal *et al.* has presented a work that reports his experience with developing tools and methods in support of this testing methodology. According to this work, model-based testing depends on three key technologies: the data model notation, the test-generation algorithm and the tools that generate supporting infrastructures for the tests [30].

Ideally, model notation would be easy for testers to understand, describe a large problem with the same easiness as it does with a small system and be understood by a test-generation tool. Also, it would be good to have an appropriate notation for requirements documents. As it is extremely difficult to achieve all of these ideals, because no model suits all purposes, there have to be several notations.

A test-generation data model defines a set of valid and invalid values that will be supplied for a given parameter while a requirements data model defines the set of all possible values for that parameter. Besides this, test-generation data model can also define combinations of values – called seeds – that have to appear in the set of generated test inputs.

The presented solution is called AETGSpec and is a simple specification notation, showing that the notation used to gather the data’s functional model can be easy to use and still produce quality test case sets. In addition, this simplicity also showed good results in terms of testers quickly learning how to write data models and generate test data. The advantages of this solution towards automatic test generation are that it has tight coupling of the tests to the requirements, it is easy for testers to learn and quick to regenerate tests in the presence of changes. Still, there are disadvantages. These are the need for an
oracle and the demand for certain development skills from testers that usually are rare in organizations. As a consequence of this pros and cons, the system to which this approach is most applicable is a system that has a data model which is sufficient to capture its behavior, meaning that there is a low complexity of the SUT’s response to a stimulus.

There are many papers presenting MBT approaches so Neto et al. presented a survey describing a systematic review performed on these approaches [31]. This formal methodology, the systematic review, is a technique to identify, evaluate and synthesize the results of relevant research on a specific question. By the year of this work (2007), it was believed that this was the first scientific survey paper on MBT approaches using systematic review. It provides various options in order to select an MBT approach for a project through qualitative and quantitative analysis of the available approaches. Namely, the analyzed characteristics are: testing level, behavior model, intermediate models, level of automation, automated support and test generation criteria.

Taking these characteristics into consideration the quantitative analyses revealed that:

- For testing level, MBT has mostly been used in System Testing (66% of all approaches) and then some approaches chose to use it in Integration Testing (22%), Unit Testing and Regression Testing;
- For automation level, the MBT approaches analyzed had most steps automated except for the first step which corresponds to the software behavior modeling;
- For supporting tools, 64% of the MBT approaches have tools to execute their steps, primarily for System Testing;
- For models used for test case generation, the most used are UML Statecharts, Class and Sequence Diagrams.

As for the qualitative analyses, it concluded that:

- For testing coverage criteria, since the available strategies are control-flow and data-flow, the best choice is control-flow because it becomes easier when it comes to evaluating full systems;
- For behavior model limitations, the used notations with better results have been UML, FSM, B Language and Z notation.

It was concluded through the experimental evaluation of the MBT solutions the existence of a lack of empirical results in MBT approaches. Furthermore, the existing approaches usually have been developed for a definite purpose and are not introduced in the industry, so this technique ends up not being frequently applied.

2.4.2 Model-based Security Testing

Model-based security testing (MBST) applies MBT in the security domain, relying on models to test if a system meets its security requirements [32]. This is considered a recent field of software testing, dedicating its attention to the systematic and efficient specification and documentation of security test objectives, test cases and test suites and also to their automated or semi-automated generation [33].
In order to be able to address the different perspectives used towards a system’s security, MBST needs to be based on different types of models [33]. Regarding this, 3 model categories were provided, each of them representing one existing perspective and with the possibility of serving as input models for test generation.

Firstly, architectural and functional models were addressed, being concerned with system requirements related to the behavior and setup of a software system. In terms of security, the focus is in locating critical system functionality in the overall system architecture and in identifying possible entry points for an attacker (by identifying security-critical interfaces).

 Afterwards, threat, fault and risk models were regarded, focusing on what can go wrong, concentrating on causes and consequences of system failures, weaknesses and/or vulnerabilities. CORAS\(^5\) is an example of an approach that falls into this category. It allows for communication, documentation and analysis of security threat and risk scenarios. Alternatively, there are other modeling tools that also fit in this category such as fault tree analysis (FTA) and cause-consequence analysis (CCA).

Finally, weakness and vulnerabilities models were addressed, concentrating on describing identified weaknesses or vulnerabilities by themselves, being based on the National Vulnerability Database\(^6\) (NVD) or the Common Vulnerabilities Exposures\(^7\) (CVE) database. Such models can be used for test generation by various MBT approaches, checking if the SUT is weak or vulnerable taking into consideration the weaknesses and vulnerabilities found in the model.

According to Schieferdecker et al., the way to generate security tests with a model-based approach can be summarized into five steps [33]. It should start by identifying security test objectives and methods, with test objectives defining the test purpose and relating goals to the testing methods that enable the objectives fulfillment. Following this, a functional test model should be designed, reflecting the expected functional scenarios of the SUT or the scenarios of the SUT usage. With this, the test generation criteria should be determined, limiting the number of the generated tests to a finite number. After this, tests can then be generated and its results should be assessed.

In addition, the DIAMONDS project\(^8\) was presented, aiming to define security fault and vulnerability modeling techniques as well as a security test pattern catalogue, to develop MBST techniques and define an MBST methodology.

So to understand which parts of the research area of MBST are and are not well-understood, Felderer et al. provided specific classification criteria – filter and evidence criteria – for existing work on this security testing approach [32]. It does so by leveraging existing MBT classification schemes and complementing them with security-specific aspects.

As for the criteria applied in order to classify the collected MBST approaches, it appears after the realization that test case generation is not tailored to security testing. It concentrates on security aspects modeling to filter security test cases and the available evidence of the approaches.

On the one hand, the filter criteria covers the specification of the system security model, the environ-

\(^5\)http://coras.sourceforge.net/
\(^6\)https://nvd.nist.gov/
\(^7\)https://cve.mitre.org/
\(^8\)https://www.fokus.fraunhofer.de/go/en_diamonds
ment security model and the explicit test selection criteria. This way it is defined a security-specific set of filters on the system model and, also, a subset of SUTs of interest for the security testing perspective. Briefly, what this criteria provide is a means to select relevant test cases.

On the other hand, the evidence criteria covers the maturity of the evaluated system, the evidence measures and the evidence level. With this it is possible to assess the industrial applicability and utility of MBST approaches and, also, the actual state in research.

Through an automatic search on some digital libraries which are considered to cover the most relevant sources in software and security engineering, and after some exclusions of irrelevant papers, 119 publications were selected. Such publications were then subjected to the classification according to filter and evidence criteria mentioned before. The results showed that every approach either specifies a model of system security or a security model of the environment and, also, that most of the approaches choose to focus on one model, instead of considering both types of models (Threat and Attack Model) at the same time. Besides this, it was possible the see that the most widely used type of security model are models of functionalities of security mechanisms and that the predominant criterion for test selection is structural coverage, being data coverage very rarely used.

From a less general – than the MBST version – adaptation of the model-based testing approach, model-based vulnerability testing (MBVT) was born with the goal of generating vulnerability test cases for web applications.

To make use of MBT techniques to vulnerability testing not only the modeling approach has to be adapted, but also the test generation computation [34]. In the MBT process positive test cases are computed to check if in the presence of certain stimuli the SUT behaves as expected. In contrast, in the MBVT process negative test cases have to be produced to check if when facing an attack the SUT lets the attacker obtain unauthorized data.

The MBVT proposed by Lebeau et al. is composed of four activities:

- Test Purposes: stands for formalizing the test purposes from vulnerability test patterns that have to be covered by the generated test cases;
- Modeling: aims to define a model able to capture the SUT’s behavior to generate consistent stimuli sequences;
- Test Generation and Adaptation: stands for, based on the artifacts produced by the 2 previous activities, automatically produce abstract test cases;
- Concretization, Test Execution and Observation: aims to translate the abstract test cases into executable scripts, to execute them on the SUT and to observe the SUT responses and compare them with the expected result. This comparison enables the assignment of a test verdict and automation of the vulnerabilities detection.

The biggest drawback of this approach that was pointed out is that it requires substantial effort to design models, test purposes and adapter. This comes as a consequence of the MBT approach, thus not being strictly a disadvantage of the MBVT approach [34].
2.5 Open Security Test Methodologies

A security methodology is the part of a process that defines what is tested along with when and where [35]. Its task is to take a complex process and reduce it into a set of simpler processes, explaining the components of those processes. As a methodology, it must explain the tests that – for each of the simpler processes – will verify their actions while they are being performed. Adding to this, the methodology must provide metrics to assure its correction and to evaluate the result of applying it.

The Open Source Security Testing Methodology Manual (OSSTMM) aims to express security in a better way for each new version that comes, providing a methodology for a thorough security test (referred to in [35] as an OSSTMM audit). With this it helps security testers to understand what makes programs safer and more secure.

The OSSTMM audit is a measurement of security at the operational level which is accurate and has no assumptions or anecdotal evidence. At the same time it is defined as:

- a methodology - designed to be consistent and repeatable;
- an open source project - allowing contributions from any security tester for performing more accurate and security efficient tests.

Regarding the OSSTMM 3.0, which is the version under analysis for this work, it defines:

1. What security testers need to know - in terms of things like security, controls and information assurance objectives;
2. What security testers need to do - regarding security testing and the things it implies like the test scope or the common test types;
3. Security Analysis - namely related to critical security thinking, recognizing the open security model, looking for pattern matching as a sign of errors and characterizing results;
4. Operational Security Metrics - explaining what the rav is (“a scale measurement of the attack surface, the amount of uncontrolled interactions with a target, which is calculated by the quantitative balance between operations, limitations, and controls”) and how to make it and some formulas related to the concepts mentioned in what the security testers need to know;
5. Trust Analysis - defining the trust concept as well as the fallacies in trust, some of its properties and rules and how to apply trust rules to security testing;
6. Work Flow - approaching the methodology flow, the test modules and a methodology;
8. Physical Security Testing;
10. Telecommunications Security Testing;

Summing everything up, using the OSSTMM is about getting a deep understanding of how everything is connected. From people to processes, systems and software, all of them are somehow connected and have to interact. When testing operations, the OSSTMM provides testers with the ability to see these interactions and the way how they affect each part of the connection.

The OWASP Application Security Verification Standard (ASVS) Project\(^9\) is, basically, a list of application security requirements or tests that can be used to define what a secure application is [36]. It provides a starting point to test the technical security controls of web applications.

There are two main goals for ASVS:

1. Helping organizations not only with developing but also with maintaining secure applications;
2. Allowing the alignment of requirements and offerings by security services, security tools vendors and consumers.

It defines not only how to use the standard, but also the existing levels of security verification. There are 3 levels: Level 1: Opportunistic - pointed at all software; Level 2: Standard - pointed at application containing sensitive data requiring protection; Level 3: Advanced - pointed at the most critical applications like the ones that perform high-value transactions and/or that contain sensitive medical data. Thus, for each level there is an associated list of security requirements that can be mapped to security-specific features and capabilities that developers must build into software.

As for the OWASP Testing Guide Project\(^10\), it shows how to verify the security of applications from another perspective. The 4th version of this OWASP document aims to be the standard guide to anyone performing Web Application Penetration Testing. It helps people testing web applications to understand what, why, when, where and how to do so [37].

Moreover, this framework highlights how important it is to test security throughout the software development process, motivating people to do so. This is considered even more relevant in the case of web applications, since they are subject to lots of users – with different intents – across the internet.

Furthermore, the OWASP Testing Framework is based in a set of activities that should take place in the following phases:

- Before development begins;
- During definition and design;
- During development;
- During deployment;
- Maintenance and operations.

The approach aims to create a testing methodology that is consistent, reproducible, rigorous and under quality control. The goal of the tests is to understand which kinds of information are exposed whether on the organization’s website or on a third party website.

\(^9\)https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project
\(^10\)https://www.owasp.org/index.php/OWASP_Testing_Project
2.6 Continuous Integration

Continuous integration (CI) is a software development practice, mostly related to how code is revised, tested and integrated into the project [38]. In this process developers frequently commit their source code changes or additions - for example a new feature - to a repository, triggering a build which is followed by tests [39]. With the test results the developer is able to see if and how his changes may or may not have affected the software, being able to fix, as soon as possible, any bugs that he was not aware of when he committed the referred changes. So the main goal of CI is to provide a quick way to report if a defect was introduced into the source code, identifying it and enabling developers to correct it promptly. As Martin Fowler affirms “Continuous Integration does not get rid of the bugs, but it does make them dramatically easier to find and remove”[11].

There is a multitude of CI software frameworks that can be used to automate both the build and the testing steps. Many of these are free tools that anyone can use like Jenkins[12], CruiseControl[13] or Bamboo[14]. Using Jenkins, Seth and Khare proposed a continuous integration and test automation implementation [39]. Their approach was to implement Jenkins in a master slave architecture for daily integration of continuous build and testing. The Jenkins server was installed on the master machine and the slave nodes were the machines where the jobs were performed. As for the build, the chosen approach was basing the build on a customized script. This script was responsible for fetching the latest code from a Gerrit[15] repository to a certain slave directory and build it, being that each of these slaves are Android devices. Then, another build step starts to run a compatibility test suite by Android, and once the tests are over, the developer teams are notified via e-mail. This notification, sent by Jenkins, states if the job was successful or not and the test results are posted on a Gerrit page. By analyzing the obtained results of executing the automated scripts and the time consumed to do so it was possible to conclude that it enhanced the employee work-hour efficiency. And also, since Jenkins is an open-source tool, it became a cost-efficient solution.

Test Orchestration is a continuous integration and continuous development framework proposed by Rathod and Surve [40]. Their motivation comes from the fact that enterprises “can no longer afford unpredictable, lengthy and ineffective release processes that barely support one update every couple of months” [40]. Consequently, Test Orchestration is a solution that accelerates and streamlines the delivery pipeline through providing a single test tracking panel which enables the manual and automated tasks to be followed. By automating all the tasks then it is possible to identify and prevent bottlenecks sooner. Moreover, this framework’s architecture focuses on:

- **Build automation** – for a product under development, every time a developer commits new code into the repository it triggers an automated build. This results not only in a software that will have less errors during its deployment lifecycle but also in having the ability to track how the latest changes affected the other software components;

---

[12]https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins
[14]https://www.atlassian.com/software/bamboo
• **Test automation** – using test frameworks to command the test execution and compare it with the expected results. Various types of tests can be performed, in this case they perform unit, integration, functional and performance testing as well as code verification and code coverage. The code is considered complete once all the tests pass;

• **Deployment automation** – continuous deployment follows the continuous integration usage, referring to the release of the software product into production once it passes the previous test automation step. It fastens software development, enabling low risk releases that can quickly be adapted to functional requirements and/or user needs, turning the release process into a faster delivery pipeline;

• **Report** – running the tests to find errors is as much as important as its results analysis. So the ultimate goal of this is to enable a more efficient test result analysis.

Based on this, Test Orchestration allows for quicker feedback based on test builds and test results analysis, improving visibility and reporting.

Summing up, continuous integration is “a set of principles that apply to the daily work flow of development teams” [41]. Its core practices are based on committing changes frequently into a repository, allowing for source code versioning, then building the software with those changes and verifying if the changes did not break anything, based on a set of defined tests. With all this, it is possible to achieve automated and fast deploys, avoiding unexpected results in a later phase of the product’s lifecycle.
Chapter 3

SQM Security Pipe

After presenting some of the works carried out in the area, this chapter will explain how the work was developed. Moreover, it is explained the environment in which the work is inserted in, the security testing tools used and the steps taken towards planning and implementing the tests.

3.1 Environment

As it was mentioned above, this section aims to briefly describe the environment that this work is a part of, starting by the product to which the tests will apply to (SQM) and the evolving into the tools used to develop the automated tests (Robot Framework) and to continuously integrate them (MPP).

3.1.1 Service Quality Manager

Service quality is defined, in a practical sense, as the customer’s perception of a delivered service. The management of this service quality stands for monitoring and giving support to end-to-end services for specific customers or classes of customers.

Service Quality Manager is a Nokia product which monitors how service quality for key services, like voice over LTE or video, is being provided and if it is degrading. More thoroughly, it provides customers with the ability to model and monitor their own services. All service-relevant information available in an operator environment can be collected and forwarded to SQM, where it is used to determine the current status of the defined services. With this, SQM simplifies service operations, providing a single point of entry for accurate and real-time service quality insight.

The current SQM version is based in a virtualized environment, with the goal of evolving to a cloud deployment in the future. The architecture of this product follows the general structure of a web application with one database server, four application servers (which will run both presentation and business servers) and one singleton server (which will run SQM Java servers), all communicating through the TCP/IP protocol.
3.1.2 Robot Testing Framework

The Robot Framework – an open-source project sponsored by Nokia – is a generic test automation framework that follows a keyword-driven testing approach. It can be used both for test automation and acceptance test-driven development. In the specific case of this work, it will be used for test automation, meaning that the tests will run with minimal or even without any level of manual actions being performed, for example someone having to constantly configure or launch the tests manually.

This framework follows a high level architecture, standing between the test data syntax layer, which has the test data above it, and the test library API layer, which has the test libraries and test tools below it.

The advantage of the Robot Framework is that it is easy to use through the RIDE editor support and can provide clear reports and detailed logs. These will, then, help with identifying the problems of the SUT in a more effective and practical way than running the tests manually. In addition, it is easy to integrate with other tools since: the test suites are created from files and directories; the command-line interface is easy to start by external tools; the output comes in XML format, thus it is machine-readable.

3.1.3 Merlin Product Pipe

Merlin Production Pipe is another Nokia in house framework. It was built to manage continuous integration (CI) pipelines, testing the software every time it is changed. The automation level provided as well as the feedback time reduction are two very important MPP features. The reason for this to be so important is, as it was mentioned before, that the sooner the developers have the test results, the sooner they will be able to fix the problems found.

In a synthesized way, the MPP production pipe works as follows:

1. A developer makes a commit to project X into an SVN repository;
2. Jenkins polls the SVN repository for changes and starts the build and unit testing of project X;
3. Build is done with MPP build scripts and the components fetched from Maven repositories;
4. The install and test cycle are initiated in MPP or MPP polls for untested builds from its release information system (RIS) and starts the install and test cycle by itself.

MPP also provides a user interface for the test environment management as well as for logs and reports, making it easier to perform these tasks. And, in addition, there are Lava Lamps, i.e., screens that show the current test results across Nokia’s offices.

---

1https://github.com/robotframework/RIDE/wiki
2http://www.slideshare.net/pekkaklarck/robot-framework-introduction
3http://subversion.apache.org/
4https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins
3.2 Threat Modeling SQM

With the purpose of analyzing and describing the attacks that SQM could suffer – in order to define the security tests that are important – we chose to follow the SDL approach as described in Section 2.3.1. From the information gathered, it will be possible to understand in which components and how the attacks can be performed and plan the tests according to this. Having a good plan will enable the designed tests to produce more accurate results, thus enabling the development of a more secure software product.

This section aims to present the results of the threat modeling that was done, namely the obtained diagrams and the threat identification performed.

3.2.1 Data-Flow Diagrams

By studying the architecture of SQM and, also, with regard to the components pointed as relevant from a security perspective, we were able to create a context diagram which allowed us to identify the main components' interactions in SQM.

From the context diagram, in Figure 3.1, it is possible to understand that SQM’s interactions focus on four components: the database, where much information needed for the SQM core to process its requests is stored; NetAct\(^5\) – a Nokia product for network monitoring – which is used as a base for SQM; SQM itself, more specifically, its core; and the User. It is important to denote that since NetAct is another Nokia product, its R&D team is responsible for its own security tests. Moreover, the DB is used as a third-party software component so in the following diagrams, which have a lower level of abstraction, we will mainly focus on detailing the SQM Core components.

In the Level 1 Data-Flow Diagram (Figure 3.2) we can see the main components of SQM’s Core and the information flow between them. The information flow is mapped according to the following steps:

- The NetAct Data Collector receives data from the network that it is in and sends it to ActiveMQ\(^6\),

Figure 3.1: Context Diagram for SQM

\(^5\)https://networks.nokia.com/solutions/netact  
\(^6\)http://activemq.apache.org/what-is-activemq.html
which will turn the received data into an event and send it to the Event Engine Manager;

- On its hand, the Event Engine Manager will receive this event, enrich it with some details coming from the DB and send it over to the Rule Engine;

- The Rule Engine processes the received event, translates it into an Alarm and sends this information to the SQM GUI;

- Having access to the SQM GUI, the user will be able to see the received alarm details and act accordingly to what he finds correct for his network.

Moving to a more detailed level, in Figure 3.3, we can see the Level 2 Data-Flow Diagram where the previously mentioned components were decomposed into their sub-components. Consequently, it is easier to understand how the information is processed and sent and how this data trades can affect the entire system. From this more detailed diagram and dividing the steps enumerated above, the information flow goes through the following steps:

- Step 1
  
  1. The NetAct Data Collector receives data from the network that it is in and sends it to ActiveMQ - which was omitted in this diagram for simplicity reasons - that will turn the received data into an event and send it to the Event Collection Framework;
2. The Event Collection Framework, upon receiving an event, sends a request to the DB to get the Service and Service Class ID which will identify the Service that will process the event. With this, the Event Collection Framework enriches the event and sends it to the JMS Queue Provider;

3. Afterwards, and considering the Service Class ID received, the JMS Queue Provider sends the enriched event to the respective Service Queue Worker which sends it to the Rule Engine Servlet;

- Step 2

1. Then, the Rule Engine Servlet fetches the rules associated with the event's Service Class ID from the DB and sends a lookup request to the Engine Factory. The response to this will be the Rule Execution Unit ID where the rule is executed, generating or not an Alarm;

Regarding the rest of the diagram, it is justified as the data input can come both from the NetAct Data Collector as well as from the User. This information flow can be described as follows:

1. The User logs into the SQM GUI and sends a Service Alarm request;

2. Receiving this request, the Service Alarm Manager sends the Service Alarm ID over to the Root Cause Analysis Servlet;

   (a) With this ID, the RCA Servlet sends a lookup request to the Engine Factory, receiving an ID for the RCA Engine which should process the Service Alarm request;

   (b) The alarm is, then, processed by the RCA Engine indicated and returns to the Servlet the output of this processing;

3. The received output, i.e. the RCA Information, is sent as a response to the Service Alarm Manager which forwards it to the SQM GUI so that the User can see the information he requested.

### 3.2.2 Threat Identification

Using the DFDs it is, then, possible to move to the SDL's second phase: threat identification. Table 3.1 summarizes the threats identified for each of the previously mentioned components.

Five of the eleven components require authentication, which represents a threat of spoofing if an attacker tries to authenticate himself to act under a valid entity in the system that does not belong to him. For the tampering threatened components, there are some different ways in which the attack attempts can occur:

- For the DB, an attacker can try to modify the DB data, namely the Service Model, the Service Instance data or the SQM configuration;

- For the Rule Engine and the RCA Servlets, an attacker can try to modify the JS and HTML templates - respectively - where the information is filled, being able to execute a piece of code on the user end;
For the Service Alarm Manager, an attacker can try to modify the received alarm information;

For the remaining components, an attacker can try to modify the collected data.

In terms of the repudiation threats, all the signaled components enable an attacker to be able to deny having performed an action, violating the non-repudiation property. For the DoS threats there are, also, some differences in the attack attempts to degrade or even deny the component's service depending on the addressed component:

- Both for the Event Collection Framework and the Event Engine, an attacker can try to overload the component with data;
- For the DB, an attacker can try to send a big amount of queries;
- For the Service Alarm Monitor, an attacker can try to deliberately trigger a big number of alarms;
- For the SQM GUI, an attacker can try to login in a brute force manner, thus sending a big load of login requests.

For the last threat type represented, i.e., elevation of privilege, three of the eleven components can be subjected to an attacker trying to gain capabilities to which he does not have authorization.

Following the SDL, the third phase would be to probe the components against the identified threats in order to verify the vulnerabilities existence and, then, proceed to their mitigation. Nevertheless, we had to choose another testing plan, due to the security testing tools available not being able to check
for the identified threats. Moreover, the tools available are suitable for a black-box testing approach, worrying about testing the tool from an outer side. Since in the threat modeling performed we focused on the inner components of SQM, in a white-box testing approach, we could not follow the initial plan. As a consequence of this restriction, the developed tests were designed by crossing what the tools were able to test and the needs felt by the product’s Software Architects which came from past experiences with reported vulnerabilities.

### 3.3 Integrated Security Tools

From a set of security testing tools recommended by Nokia, and based on the types of tests we wanted to perform, we chose three tools. This section aims to briefly present these tools, the features that were used and, also, the reasoning about these choices.

#### 3.3.1 Arachni - Web Vulnerability Scanner

Arachni\(^7\) is a free and open source web application security scanner framework. It is reported to have a high accuracy and a low false positives percentage, making it a good tool with little associated costs (when compared to the non-open source options) and in constant update.

In order to scan an application and look for web vulnerabilities the user is able to define several testing profiles, choosing what he wants to verify and how he wants to do so. Namely, and among other things, a user can choose:

- **Active checks** - which actively engage the web application through its inputs;
- **Passive checks** - which passively collect data;
- **Plugins** - that extend the system’s functionality and provide more information about the application;
- **Reporters** - that store and present scan results in several formats like json, xml, yaml or txt.

\(^7\)http://www.arachni-scanner.com/
Arachni can be used through its web user interface and/or its command-line interface. For the purpose of this work, the web UI was used to define the testing profiles while the command-line interface was used to launch the tests and generate the respective reports.

### 3.3.2 Nessus - Vulnerability Scanner

Nessus\(^8\) is one of the most widely used and trusted vulnerability scanners. Its popularity comes from it being able to support a wide range of network devices, OSs, DBs and applications. Moreover, it has multiple scanning, configuration and compliance options - which are chosen upon the policy creation - and is constantly being updated whenever new known vulnerabilities are announced.

From the several types of scans users can create, for example Basic Network Scans or Bash Shell-shock Detection, we used the Advanced Scan type, which is the type of scan that can be configured freely and without any of the underlying recommendations of the other scan types. Once the scan type is chosen, the user can configure the scan settings, namely its name, policy and target IP(s).

Regarding the policy, there is also a variety of pre-configured options that the user can choose from to best fit its needs. With the policy definition plugins can be chosen, i.e., the vulnerability types that the scan will cover as well as any needed authentication to the scan targets can be configured. For SQM Security Pipe and since Nessus also has the option of using a customized policy, we used the recommended Nokia security policy, being that the only thing that we needed to configure was the authentication to the several virtual machines (VMs) that exist in the SQM system.

As in the Arachni approach, we used Nessus web UI to configure the policy and we used its REST API to create and launch the tests, generate the reports and partially validate the results.

### 3.3.3 Nmap - Port Scanner

Nmap\(^9\)(Network Mapper) is a tool for network exploration, being open source and widely used for security auditing. It is mostly used to verify which are the available hosts on a network, as well as the services being run on those hosts and their reachable ports.

Using Nmap's command-line interface it is possible to launch various scan types with multiple configurations. For the SQM Security Pipe case we opted for a TCP scan, a UDP scan and an IP Protocol scan, which will be described in Section 3.4. With the scans output the user can avoid having open ports that he is unaware of or having suspicious services running that can indicate malicious activity.

There is also a GUI for Nmap called Zenmap, but since the scan options are simple to use through the command-line interface, this GUI was not used.

---

\(^8\)https://www.tenable.com/products/nessus-vulnerability-scanner
\(^9\)https://nmap.org
3.4 Test Development

The Nokia DFSEC (Design For Security) is a company process for developing secure products and systems, containing controls and deliverables to help product lines with creating secure products. It aims to proactively make Nokia products secure from the start of their development. As a part of this process, SQM, as other Nokia products, is tested for security vulnerabilities using the set of previously mentioned tools. These tests ensure that no known security vulnerabilities are included in each product release.

SQM has a build pipe based on MPP (Section 3.1.3), so the developed tests were integrated into this pipe in order to provide earlier feedback on the detected vulnerabilities. In a more detailed manner, as depicted in Figure 3.4, what happens is that every time a developer checks in new source code into the SQM repository it starts the MPP cycle, initiating a new product build. Afterwards, that build is installed in a lab where a set of automated functional tests are run assuring that the newly developed work did not introduce any flaws into the software. What this work added was a new set of automated tests to be run after the functional tests, creating the SQM Security Pipe. The difference between the functional and the security tests integration is that the security tests are only run once on the first build of the day and not upon each build. This happens due to the time required to run these tests and given that we decided that once a day would be enough to keep up with the vulnerabilities reported and detected.

The developed tests will be explained in further detail in the following sections. Namely, it will be explained how the tests were structured using the Robot Framework (Section 3.1.2) and their respective security testing tools.

3.4.1 Web Vulnerability Tests

Following the robot framework structure, test data is defined in .txt files through the RIDE editor. A group of these files together, being each a test case (a robot), creates a test suite.

For the web vulnerability test suite we developed 5 robots each making use of a testing profile created with Arachni’s web user interface. The difference between the testing profiles is the security checks performed, having both different active and passive checks. Active checks are the ones that actively engage the web application via its inputs, whereas the passive checks are the ones that passively collect data as an extra form of information for the test output.
Addressing the 5 robots, we have:

1. **SQL Injection**
   - **3 Active Checks**: SQL injection, Blind SQL injection (differential analysis) and Blind SQL injection (timing attack);
   - **1 Passive Check**: Interesting responses;

2. **XSS**
   - **8 Active Checks**: XSS, DOM XSS, DOM XSS in script context, XSS in HTML element event attribute, XSS in path, XSS in script context, XSS in HTML tag, CSRF;
   - **1 Passive Check**: Interesting responses;

3. **NoSQL Injection**
   - **2 Active Checks**: NoSQL injection and Blind NoSQL injection;
   - **1 Passive Check**: Interesting responses;

4. **Code Injection**
   - **3 Active Checks**: Code injection, Code injection (php://input wrapper) and Code injection (timing);
   - **1 Passive Check**: Interesting responses;

5. **File Inclusion**
   - **2 Active Checks**: File inclusion and Remote File Inclusion;
   - **6 Passive Check**: Interesting responses, Backdoors, Backup directories, Backup files, Common directories and Common files.

We do not describe all the tests as that would be tedious. However, we present an example with the SQL Injection test case, aiming to clarify how the active and passive checks work. An SQL injection attack consists in inserting an SQL query into an application via its client’s input data, affecting the execution of predefined SQL commands. Considering this, an active check would be to search for the application’s input entry aiming to, through there, send various SQL queries, waiting for its response and verifying if it contains an error known as a result of a successful SQL injection attack. A passive check would be to analyze the network traffic throughout the interaction with the application, pointing out the potentially interesting information it was able to get from all the HTTP requests and responses.

Before executing any of the keywords of a test case, the robot checks if the `Setup` keyword has any actions to perform. Firstly, for the test suite, we created a keyword named `prepare lab dir` which checks if a directory named after the lab name (for example: CloudLab117) exists and has the permissions necessary to be read and written by the user running the tests. If the directory does not exist or the permissions are not set properly, then this keyword creates the directory and/or sets the right permissions. This is important due to the fact that when MPP runs a set of tests it creates a user solely for...
that purpose. Once the tests are done, MPP copies certain folders to fetch the results and then destroys both the user and the data he created. If the user does not have the right permissions set to access the directory and run the tests, then the test fails. After running the test suite setup, each test case will run its own setup, using the `testcase dir`. This keyword follows the same logic as the `prepare lab dir`, but instead of creating a directory for the lab it goes into the lab directory and creates a directory named after the test case (for example: SQLinjection).

Once the setup of the test case is done, then the robot starts executing the rest of the keywords. The structure of the test case should be as abstract as possible so as Figure 3.5 shows the test case is simple to understand even if a user is not familiar with tests automation.

When in an MPP testing laboratory, there is a Python file defining some environment variables that are used throughout the functional tests. Taking advantage of that, each robot can get the target IP ($SDVCPF LWASACCESSADDRESS) for the web vulnerability test suite from the referred file. Having that and the SQL Injection profile that had been previously configured and saved, Arachni can be executed. The `execute arachni` keyword uses the arachni command-line interface to launch arachni with the given arguments and waits for its execution to be over. Upon the end of arachni's execution the robot uses the `get preReport path` keyword to fetch the complete path of the report generated in a .afr file and then passes this as an argument to the `generate arachni report` keyword. This keyword will use, once again through the command-line, an arachni plugin which is able to create reports in different file formats from the .afr report. Namely, it creates a .html report that is saved (through the `save Arachni report` keyword) to be analyzed later by the person managing the tests and a .json report which will be used for the robot test case validation.

As a final step, the `get Report results` keyword divides itself into two main actions: it goes through the .json report using a command-line json parser called jq\(^\text{10}\) to validate the test results and it retrieves only some of the information provided in the report in order to create a customized short report to be sent as the output of the robot in the MPP results. The short report is important because it helps to minimize the amount of information that the developer needs to go through in order to understand and correct the vulnerability detected. Also, in relation to the validation, it focuses on verifying if there were high or

---

\(^{10}\)https://stedolan.github.io/jq/
medium severity vulnerabilities detected. If there are, then the robot checks in the blacklist created for this test suite if the vulnerabilities are already reported or not and in case they are not, when producing the short report, these are stamped with a “[Rejected]” stamp. In case they are already reported, then they are stamped with “[Accepted]”. The blacklist, as the following example in Figure 3.6 demonstrates, is a .txt file which contains a field indicating the last product version to which it was updated which is followed by a list of the desired vulnerabilities. This list contains the vulnerability name and the vector url which enables the user to differentiate between vulnerabilities with the same name. As a final step, the robot goes through the short report and if there any vulnerabilities stamped as rejected the test fails.

Every test case in this suite follows this structure, being that the main difference between them is the testing profile argument which differs from test case to test case, as explained in the beginning of this section.

When all the test cases of the test suite are done, independently of their result, a **Teardown** keyword is always run, assuring that files which are not needed, i.e., the .afr and .json reports, are erased from the lab. Also, only the three most recent .html reports and short reports are kept for each test case in order to manage the free space in the machine.

### 3.4.2 Vulnerability Tests

The vulnerability test suite contains 2 robots, one to perform an authenticated scan and another to perform an unauthenticated scan, both using Nessus. Despite of not having a big number of test cases, the used policy has a big range of selected security checks, meaning that a wide range of known vulnerabilities are covered.

As in the web vulnerabilities test suite, the only difference between the test cases is the testing profile used, but for Nessus the profile is called policy. For these tests the recommended Nokia security policy was used, having the same checks for both test cases but having authentication credentials configured for one and not for the other. This enables the testing manager to compare both results and conclude how authentication helps SQM to be more secure.

Regarding the test cases structure, the same **Setup** and **Teardown** keywords used in the previous test suite are used for this one. The main structure of the test cases was developed taking into consideration both what we wanted to achieve with this test suite and also what the Nessus REST API used required.

Firstly, we needed to create a keyword (define scan targets) which could dynamically define the targets of the vulnerability scan. Although some of the target IP's were defined in the file referred in the previous section, not have all the targets were there. In spite of this, we found a way of dynamically cre-
ating a complete targets list using that file and the knowledge we have on SQM’s architecture. From the variables definition file, the robot fetches the first target IP from the $SDV_DB_HOSTNAME_IP variable. Since we know that the DB is the first node, the ASs are the following 4 and, the fifth is the singleton node, we increment the DB IP following that order, ending up with a list with these 6 IP’s. As a final add, the robot fetches the load balancer’s IP from the $SDV_CPF_LB_WAS_ACCESS_ADDRESS variable.

Having the targets, the set nessus args keyword defines the Nessus remote machine’s IP that will be used, the port for the connection as well as the Nessus’s user and password. On the contrary to the previous test suite where the security testing tool used was installed in the TEX, for this case and due to licensing reasons, we had to use Nessus in a remote machine and retrieve the results to the TEX. Since both the Nessus machine and the lab were located in Nokia’s Finland server, this did not represent a problem in terms of communication.

After having set the arguments, the robot sends a token request (through the generate nessus token keyword) in order to obtain authorization to perform the following requests. Since we had two Nessus machines available, if the default Nessus machine is unavailable the robot will detect this through the response sent for the token request and will redefine the Nessus arguments to use the other Nessus machine. If both machines are unavailable the test fails immediately. On the other hand, if a Nessus token is successfully received the robot moves to the get scan id keyword. What this keyword does is it creates a new scan defining the policy that will be used, the scan name, its targets and the e-mail address that will receive a summed up report from the Nessus machine once the scan is over. The output of the request will be the created scan’s id, that will be retrieved by the run nessus scan keyword so that the scan is launched.

When the scan is over the get found vulnerabilities keyword starts. In first place it sends a request to the Nessus machine to retrieve the scan results and filters the response with the jq json parser. This filtering aims to get only the critical and high severity vulnerabilities for each host (i.e. each target). So, for each host the robot creates a list of ids from the found vulnerabilities. Following the same blacklist logic as in the web vulnerability test suite, it verifies if the vulnerability found is accepted or not, performing the match through the id of the vulnerability. Gathering this information for all the vulnerabilities, a short report is built stating the name of the vulnerability, the hosts where it was found, the solution suggested by Nessus to fix it and the stamp indicating if the vulnerability is accepted or rejected.

By the end of that keyword’s work, Nessus’s original pdf report is downloaded from the Nessus machine – using the get nessus pdf report keyword – and stored in the TEX to be analyzed later. Finally, validation is performed by the robot passing the test if there are no rejected vulnerabilities found or failing the test otherwise.

3.4.3 Port Scanning Tests

The port scanning test suite has 3 test cases: a tcp scan test case, a udp scan test case and an IP protocol test case. In the same way as the previously presented test suites, it also uses the Setup keyword to assure all the results are stored and organized. After everything is set up, the get node infos
Figure 3.7: TCP Scan - test case example

keyword is used to create the scan targets list, as explained for the `define scan targets` keyword in the vulnerability test suite but without adding the load balancer IP. So what we get is a dictionary where the (key,value) pairs are mapped as (node type, node IP), resulting in a dictionary with 6 entries: (DB, DB IP), (AS_1, AS_1 IP), (AS_2, AS_2 IP), (AS_3, AS_3 IP), (AS_4, AS_4 IP) and (SGLT, SGLT IP).

With all the targets defined, the core of the test starts, being based on a for cycle. This cycle, as Figure 3.7 shows, is initiated on line number 9 of the test case, going through each of the referred nodes and performing the following actions:

- Nmap is executed (called in line 15 of the test case), using its command-line interface, with a command of the form: `nmap <scan options><target IP>`. As expected, the scan options depend on the type of scan intended, i.e. depending on the test case we are running;

- Using the `.xml` report generated by Nmap once it ends the scan, the robot saves the report, as line 17 shows;

- The `parse ports info` keyword, called in line 19, gets the report and parses its information, obtaining a new dictionary containing (key, value) pairs mapped as (port id, service name). These pairings represent the open ports found and the services being run on those ports;

- Based on SQM’s architectural knowledge, a list was set - for each node type - to map the expected open ports and their respectively running services, for example `{service.port.AS}`. With that list, the robot simply uses the `service Port Dictionary Creation` (lines 21 to 23) to parse it and build a dictionary similar to the one obtained after parsing the scan results;

- Apart from the expected open ports and services list, there is also one list per node type for each scan type. This list only represents the ports that are expected to be open for each protocol, i.e.,
TCP and UDP. For the IP protocol scan, this list represents the services expected to be running. Using these 3 lists validation is performed, in lines 24 to 26, firstly verifying if the open ports found are expected to be open and, if so, if the services running on them are also the expected ones. In case one unexpected open port is found or an unexpected service is found running the test fails. This information is stored in a short report generated for each scan stating - once again for each node - the unexpected ports and/or services found.

It is important to mention that since the UDP scan took more than 24 hours to run this cycle we had to restrict it to run the scan only on one target per test suite run, instead of running on a list of targets as the other scans. The target is randomly selected from the targets list, and since the list has 6 targets, several scans are run for all the nodes by the end of one month.

To finish the test case, the short report is saved and the robot checks it in order to know if there were unexpected ports found. If there were no unexpected results, then the test finishes with a passing verdict.

### 3.4.4 Final Work Summary

Summing up the developed work, we designed and fully implemented 3 test suites, each with their specific test cases. Namely, as Table 3.2 shows, for the web vulnerability test suite, each of the the 5 test cases has a main structure of 10 lines which define the test in a high level of abstraction. While for the vulnerability test suite we have 2 test cases with a main structure of 19 lines, and for the port scanning test suite we have 3 test cases, with a main structure of 37 lines each. In a lower abstraction level, presented in Table 3.3, each of the referred web vulnerability test cases has a total of 10 keywords, which adding to the main test case structure represent a total of 104 lines. Regarding the vulnerability test cases, with 19 keywords each, we got a total of 285 lines and for the port scanning test cases, also with 10 keywords each, we got a total of 184 lines. An example of one of the developed tests is presented in Appendix A.

For future maintenance of this work, a Confluence\(^{11}\) – a team collaboration wiki software – page was created explaining the developed work and some of the details regarding the setup of the environment.

---

\(^{11}\)https://www.atlassian.com/software/confluence

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Test Cases</th>
<th>Lines per Test Case (Main Structure)</th>
<th>Keywords per Test Case</th>
<th>Lines per Test Case (Total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Vulnerability</td>
<td>SQL Injection</td>
<td>10</td>
<td>10</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>XSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NoSQL Injection</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Code Injection</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>File Inclusion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vulnerability</td>
<td>Authenticated Scan</td>
<td>19</td>
<td>19</td>
<td>285</td>
</tr>
<tr>
<td></td>
<td>Unauthenticated Scan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port Scanning</td>
<td>TCP Scan</td>
<td>37</td>
<td>10</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>UDP Scan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP Protocol Scan</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Developed Test Suites and Test Cases Summary
<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Test Case</th>
<th>Keywords</th>
<th>Keyword Lines per Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Vulnerability</td>
<td>SQL Injection</td>
<td>- prepare testcase dir (9 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XSS</td>
<td>- execute arachni (10 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NoSQL Injection</td>
<td>- get preReport path (4 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Code Injection</td>
<td>- generate arachni report (9 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>File Inclusion</td>
<td>- save Arachni Report (13 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get Report results (22 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- parse results (4 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- check if in blacklist (13 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get sqmproduct version (4 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- clean reports (6 lines)</td>
<td></td>
</tr>
<tr>
<td>Vulnerability</td>
<td>Authenticated Scan</td>
<td>- prepare testcase dir (9 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- define scan targets (15 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get node infos (15 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- set nessus args (5 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- generate nessus token (5 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- check availability and retry (11 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get scan id (23 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get folder id (15 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- run nessus scan (3 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get found vulnerabilities (39 lines)</td>
<td>266</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- wait for scan completion (12 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- sum up results (31 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- check if in blacklist (13 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get sqmproduct version (4 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- add to host plugin (3 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- parse plugin_out (11 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get nessus pdf report (6 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- wait for file (11 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- clean up (35 lines)</td>
<td></td>
</tr>
<tr>
<td>Port Scanning</td>
<td>TCP Scan</td>
<td>- prepare testcase dir (9 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- get node infos (15 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- prepare node dir (3 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- execute nmap (10 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UDP Scan</td>
<td>- save Nmap report (18 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- parse ports info (33 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP Protocol Scan</td>
<td>- get and split report (8 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- service Port Dictionary Creation (5 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- validate Reports results (34 lines)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- port range extra check (12 lines)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Developed Test Cases and Keywords Summary
where it is inserted in. An example of this is presented in Appendix B.
Chapter 4

Evaluation

In the present chapter, we will explain why the previously existing solution could not be used anymore, how the developed work improved the previous solution and, also, which are the advantages it brings to SQM’s development. Section 4.1 will go into the previous solution (STAS) and then Sections 4.2.1 and 4.2.2 will go into the qualitative and quantitative evaluation of the work, respectively.

It is important to denote that we felt the need to introduce STAS in this section due to the comparisons made in Section 4.2.

4.1 Previous Solution - STAS

STAS is a security tests automation framework developed at Nokia with the purpose of creating one interface that would facilitate the use of several security testing tools as well as making these tools configuration and results retrieval easier. The most recent version of this tool was released in 2013 and could handle the execution of four test case types: port scanning using Nmap; vulnerability scanning using Nessus; robustness testing using Codenomicon; web vulnerability scanning using Acunetix.

The functioning of STAS works around four .xml files from which it is able to fetch the tools’ configuration and the tests’ setups. More thoroughly, these files are:

- *sut.xml* - a file defining the host details, with entries in the form `<host IP>` `<host name>` `<host type>`;
- *engine.xml* - a file defining the test tools locations and parameters, with entries in the form `<host IP>` `<host port>` `<user>` `<password>`;
- *testcases.xml* - a file defining the specific arguments for each testing tool, with entries in the form `<test case name>` `<tool name>` `<tool arguments>`;
- *type_<X>_*.xml* - a file defining host type specific parameters, with the entries form depending on the test case. For example, for a port scanning test it would define the expected open ports, while for a vulnerability scan it would define the expected report.
The tests would be automatically run once a week providing delta reports for all the tools, comparing the results with the scan results from the previous week.

In spite of initially having tried to use this framework to implement the new set of security tests, there were several difficulties that made us think of another solution:

1. According to the STAS architecture, the four .xml files mentioned above have a fixed content regarding the configurations of the tools and of the tests;

2. STAS is prepared to use Acunetix for the web vulnerability scans but Acunetix can only be run on Windows machines. Since the current test machines are all Linux machines, web vulnerability scans using STAS could not be performed anymore;

3. STAS is not updated since 2013, so there were compatibility problems that had to be solved. Namely, STAS was configured to use Nessus through an XML-RPC API which, for the latest Nessus releases stopped being supported and was replaced by a REST API which STAS is not ready for;

4. STAS was set up to automate tests, but it didn’t regard the labs installation. As a result, when test runs were needed, someone would have to install a lab manually with the latest SQM version, so that the tests could be run.

Regarding the first problem, the engine.xml and the testcases.xml files represent a low impact, since the tools are installed on a specific TEX which will unlikely change. On the other hand, the tests performed on MPP are run on virtual machines that can vary and this represents a high impact, since there would have to be someone who would have to constantly update the sut.xml file in order for the tests to run on the appropriate target. This and the second problem arose from the fact that STAS was created and planned to be used with physical machines. SQM, in its present version, is a virtualized application where the number of VMs can grow so STAS became difficult to use under SQM’s scope.

As for the third problem, it is aggravated by the fact that SQM’s architecture changed a lot over the last two years and STAS was not updated according to that evolution, so most of the configurations became obsolete. Finally, in relation to the fourth problem, it also represented a high impact in terms of time. So, due to all of the effort it would require to update STAS before being able to design and develop the desired tests, this idea became unfeasible.

4.2 Developed Solution

Once the tests were fully developed and integrated we focused on evaluating the final results obtained from the SQM Security Pipe, presented in Chapter 3. In order to understand which were the gains of this new solution, we performed a qualitative evaluation as well as a quantitative evaluation.
4.2.1 Qualitative Evaluation

In terms of qualitative evaluation, we started by comparing SQM Security Pipe with other solutions implemented by some Nokia teams around the world. Enumerating the solutions we found available, we had:

1. *Nokia India* - achieved a medium level of automation with Nmap and Nessus scripts. The first disadvantage that we identified was that all the scripts require the effort not only of being launched manually but also of having to update the scan arguments each time a scan needs to be launched. The other considerable disadvantage identified was that the referred scripts only configure and launch the scans, leaving the results analysis to be carried out manually, thus representing a big time effort.

2. *Nokia Hungary* - designed 5 types of security scans: port scanning, vulnerability scanning, robustness testing, DoS testing and web vulnerability scanning. The biggest problem of this solution is that there is no automation or integration achieved. The designed scans are mostly run upon each product release and all of the steps have to be performed manually, from the configuration of the tools to their results analysis. Consequently, this solution was found to be even more time consuming than the *Nokia India* solution.

3. *Nokia Poland* - achieved test automation for vulnerability scanning, port scanning, web vulnerability scanning and robustness testing using Nessus, Nmap, Arachni and Codenomicon, respectively. Vulnerability scanning automation was achieved using the Nessus REST API continuously integrated with Jenkins. The problem of this for SQM’s development case is that Jenkins is solely used for build automation, while continuous integration is assured by MPP. Using Jenkins for continuous integration would mean that the security cycle would not be watched and maintained as the rest of the existing testing cycles. In addition, result analysis was left to be performed manually, which - as it was mentioned before - is very time consuming. Nmap automation is also run by a Jenkins job, but validation is done using a Python script which checks for port openings against the communication matrix. Still, the script has to be run manually in order to get the validation results. For Codenomicon, automation was achieved executing it through the command-line and using a testplan which was previously configured through the Codenomicon GUI. As for the Arachni case, it is run through the command-line and – each time a scan is launched – all the scan specifications have to be passed as arguments for the Arachni executable. Both Codenomicon and Arachni have no automated validation associated with them, so the scans have to be launched and validated manually.

All in all, most of the developed solutions have a low level of automation, having many steps which require manual intervention and, also, do not save any effort in terms of results validation. So, comparatively to our solution, we believe we have achieved both the most automated as well as the most effort saving option available.

In comparison to STAS, the SQM Security Pipe has multiple advantages:
<table>
<thead>
<tr>
<th>Security Tests Metrics</th>
<th>Before STAS</th>
<th>With STAS</th>
<th>With SQM Security Pipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Cycle Run Time</td>
<td>120 hours (5 days)</td>
<td>36 hours (1 and a half days)</td>
<td>9 hours</td>
</tr>
<tr>
<td>Security Cycles Run</td>
<td>4 per year (2 per major release)</td>
<td>52 per year (1 per week)</td>
<td>365 per year</td>
</tr>
<tr>
<td>Results Analysis Effort</td>
<td>2 days</td>
<td>2 days</td>
<td>1 day</td>
</tr>
<tr>
<td>Feedback Time</td>
<td>7 days</td>
<td>2 days</td>
<td>1 day</td>
</tr>
<tr>
<td>Opened Prontos</td>
<td>unknown</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Security Cycle Metrics - before the STAS solution, with the STAS solution, with the SQM Security Pipe solution

- It does not need a file defining all the target IPs. The robot generates the targets list dynamically for each scan, enabling the tests to be performed on any desired testing lab without having to spend time redefining IPs and host types;

- Tool configurations are all performed by the robot, eliminating the need for any fixed files to exist;

- It has a more effective way of validating results, due to the automated validation also performed by the robots;

- Feedback is faster, thanks to the MPP integration, allowing the developer to easily know where and/or how he created a security problem. Even when approaching final release dates, changes to SQM can be made and the security cycle will ensure that there were no new security issues created, avoiding the risk of having last minute problems to solve when releasing a new product version.

### 4.2.2 Quantitative Evaluation

As for quantitative evaluation, there are some important metrics that show the gains that SQM Security Pipe brings, when compared to what existed before the STAS solution and with the STAS solution. As Table 4.1 shows, SQM Security Pipe has brought many advantages to SQM's development. Firstly, the security cycle time reduction enables the feedback time to the developer to be shortened by a day, so any security issues that may appear are flagged and assessed within a short period of time. In spite of this, it is important to take into consideration that STAS has a bigger cycle time due to the robustness tests, which take an average of 5 hours per test case. However even if robustness testing was implemented, SQM Security Pipe’s solution would still have a shorter feedback time, mostly due to the tests integration on MPP.

Secondly, the number of security cycles run has evolved from being run 2 times per major release (being compliant with the mandatory scans) to being run once a week, providing weekly results. And, as a final evolution with SQM Security Pipe, the scans are now run once a day, resulting in faster feedback and less effort on the development team part.
In addition, even though we were not able to quantify these, we had some initial errors that lead to test failures. Namely, we had two main situations: failures detected upon the MPP integration and failures detected upon the robot integration. Regarding the first type of failure, it was caused by lab installation problems not related to the tests themselves. The second type was caused by issues which were found and solved during the tests development, like unexpected open ports found because the communication matrix was not updated in time and/or like obsolete libraries being used that are known to have vulnerabilities.

Ultimately, all the improvements with security cycle runs, analysis effort and feedback time result in a lower number of opened prontos. This is an important benefit, since a security pronto takes an average time of 40 hours to be solved. Before SQM Security Pipe, and for the last two releases, 3 prontos were opened, representing 120 hours dedicated to solving problems that could take less time to solve if they had been reported sooner in the development phase. A big advantage of having scans running daily is that during the next day the developer is likely to still be working on the same thing and if a security issue is reported from the day before, then it will be a lot easier to remember the details of what was changed and could have caused the problem. Furthermore, less time solving an issue means less financial costs, being that STAS had already represented a 75% costs reduction. With SQM Security Pipe this percentage is even higher, although there is no official data to state the concrete reduction represented by our solution yet. Despite this, since SQM Security Pipe was implemented, no security prontos have been opened.

### 4.2.3 Stakeholder Evaluation

After finishing the implementation phase, we did an internal presentation at Nokia on the SQM Security Pipe. SQM’s developers, tester, software architects and line manager were part of our audience, as well as a few people from other Nokia teams, which are interested in having something similar to our solution for their own products. The presentation was also recorded and forwarded to two people at Nokia Finland and one person at Nokia India who are security leads of their products and who have helped us throughout the development of this work, namely sharing their solutions that we have briefly described in Section 4.2.1.

In order to justify not only the aim but also need for this work, Engineer José Ramos started by introducing the Nokia DFSEC and the previously existing solution. Explaining, also, what SQM Security Pipe does, how it is inserted into SQM’s day-to-day development and why it is advantageous.

Then, I started by introducing the integrated security tools used, explaining what they do, why and how we use them. Moving into explaining how these were integrated into the robot framework tests and, afterwards, into MPP. Lastly, future developments were discussed leaving time for some questions.

By the end of the presentation, the feedback was very positive, pointing out that SQM Security Pipe saves a lot of work for the ones that had to configure the tests and reduces the worries about last minute security issues showing up close to a release date. Also, some people mentioned how important this  

---

1 Pronto is a Nokia’s tool for managing faults found in its products. Every time a fault is found, a ticket, called pronto, is opened describing the problem found and alerting the responsible team. Once the problem is solved, the pronto is closed.
work is, due to the fact that security is something that companies are starting to look at with closer attention in the last few years.
Chapter 5

Conclusions

Finally, Chapter 5 sums up how the initial goals were fulfilled in Section 5.1 and, in Section 5.2, points out what can still be improved.

5.1 Achievements

With the present work we were able to design and develop a set of security tests that are currently being run in a continuous integration build pipe for a Nokia product.

The first approach of designing the tests from SQM's threat modeling (Section 3.2) could not be used due to the tools we had available. As a consequence, the tests could not be planned according to the fields of confidentiality, integrity, availability, authorization and non-repudiation as we had initially aimed. In spite of this, threat modeling was an important step to get a better understanding of SQM's architecture, as well as how the interactions between components can affect the system as a whole. Considering this, the tests were designed using the tools available in the way that would best fit SQM's needs in terms of the types of vulnerabilities that could threaten it.

Using Robot Framework to automate the tests and, then, using MPP to integrate them the SQM development team now has a way of knowing if the new features being introduced into the product also introduced security problems. Adding to this, the feedback is given automatically for each day and it is synthesized so that it is easy to understand where the problem was found and what it may represent.

Furthermore, in spite of the tests having been developed with a specifically targeted product, with few adjustments they can be easily used for other products. Namely, what needs to be redefined is the way targets are assigned and the specific expected results that should be compared to the ones presented by the tools output.

Considering all of the points above, all in all the main goal of designing, developing and integrating a set of security tests into Nokia's build pipe was achieved successfully.
5.2 Future Work

The developed work left room for, in the future, extending the test suites to other types of security tests.

Due to timing and licensing constraints we were not able to develop the fourth type of security test planned, a robustness test suite using Codenomicon. It will have to be evaluated if this is possible, since, from previous experiences using Codenomicon manually, its test cases are heavy in terms of time consumption, which can be a disadvantage towards continuous integration.

Adding to this, Arachni's authentication options were not fully developed when these tests were implemented. As a result, the web vulnerability tests are being performed without any authentication, leaving a lot of pages (inside SQM) without being scanned. So, as this scanner evolves, it would be a great advantage to study how to use the authentication option in order to have a better web vulnerability scanning coverage.
Bibliography


Appendix A

Test Example

This appendix shows a simplified example of one of the designed and developed tests, demonstrating how the robots were structured.

Since this is a simplified example, some variable and keyword definitions were omitted. It is also important to explain that all the lines starting with a # are comments left in the code, in order to facilitate its reading.

The chosen example is the authenticated vulnerability scan. The test suite starts by the definition of its settings, namely some information about its documentation and the definition of the default setup, which is the first thing to be run.

Then, the variables are defined, moving to the test cases. Like the test suite, the test case starts by defining its settings, in this case the test case tags and its setup. Then the test is structured, using the keywords as explained in Section 3.4.2. And, finally, after structuring the test case, the implementation of the keywords is presented.
*** Settings ***
Documentation
Nessus Full Reports are available at
*<scantype>_NessusReport_<year><month><day>.pdf*
Suite Setup
prepare lab dir  ${NESSUS_REPORTS}

*** Variables ***

${NESSUS_REPORTS}  ${TEMPDIR}/NessusReports
${NESSUS_USER1}  XXXX
${NESSUS_PASSWORD1}  YYYY
${NESSUS_USER2}  XXXX
${NESSUS_PASSWORD2}  YYYY
${NESSUS_IP1}  10.91.215.254
${NESSUS_PORT1}  :8834
${NESSUS_IP2}  10.9.212.124
${NESSUS_PORT2}  \  # does not exist

${NOKIA_AUTHENTICATED_POLICY}  Nokia_authenticated_local_vulnerability_scan
*** Test Cases ***

Authenticated_local_scan

[Tags] nessus_authenticated

[Setup] prepare testcase dir AuthenticatedLocalScan ${NESSUS_REPORTS}

#define scan targets
define scan targets

#get scan date

${year} ${month} ${day} = Get Time year,month,day

#set default values for Nessus IP, USER and PASSWORD

set nessus args ${NESSUS_IP1} ${NESSUS_PORT1} ${NESSUS_USER1} ${NESSUS_PASSWORD1}

#generate nessus one time token

${tryAgain} = generate nessus token

Run Keyword If ${tryAgain} generate nessus token

${scan_id} = get scan id ${TOKEN} ${NOKIA_AUTHENTICATED_POLICY} SQM_${lab_name}_${test_name}_${year}${month}${day} ${targets_vm1_to_6_and_lbwas} mariana.paulo@nokia.com

#run scan based on given policy run

nessus scan ${TOKEN} ${scan_id}

#get vulnerabilities found in the scan and sum up results if critical or high vulnerabilities are found

${vuln_found} = get found vulnerabilities ${TOKEN} ${scan_id} ${year} ${month} ${day}

#get pdf get nessus pdf report

${scan_id}

#final validation

${results} = Run Keyword If ${vuln_found} OperatingSystem.Get File ${results_summary}

Run Keyword If ${vuln_found} Should Not Contain ${results} Rejected Found vulnerabilities that aren't addressed in the blacklist! Marked as "[Rejected]" in the summed up report.

[Teardown] clean up ${TOKEN}
*** Keywords ***

**define scan targets**

#get AS, SGLT and DB

`${node_infos}` = get node infos `${SDV_DB_HOSTNAME_IP}`

#get lbwas ip

`${rc}` = Run And Return Rc And Output traceroute`${SDV_CPF_LB_WAS_ACCESS_ADDRESS}`

`${ip}` = Fetch From Right `${out}` `${SDV_CPF_LB_WAS_ACCESS_ADDRESS}${SPACE}`

`${ip}` = Fetch From Left `${ip}`

#add lbwas to dictionary

Set To Dictionary `${node_infos}` LBWAS `${ip}`

#create targets list

`${targets}` = Get Dictionary Values `${node_infos}`

Log List `${targets}`

`${targets}` = Convert To String `${targets}`

`${targets}` = Remove String `${targets}` [ ] `u`

Log `${targets}`

Set Test Variable `${targets_vm1_to_6_and_lbwas}` `${targets}`

**set nessus args**

[Arguments] `${ip}` `${port}` `${user}` `${password}`

Set Test Variable `${NESSUS_IP}` `${ip}`

Set Test Variable `${NESSUS_PORT}` `${port}`

Set Test Variable `${NESSUS_USER}` `${user}`

Set Test Variable `${NESSUS_PASSWORD}` `${password}`

Log Many `IP:``${ip}` \n `PORT:``${port}` \n `USER:``${user}` \n
**check availability and retry**

[Arguments] `${beforeGettingToken}`

`${isNessus2}` = Run Keyword And Return Status Should Be Equal As Strings `${NESSUS_IP}` `${NESSUS_IP2}`
#command is different because one nessus server has a specific port for nessus and the other doesn't. Since ports can be pingged, nmap will check if the port is open or not

$\{\text{command}\} = \text{Set Variable If } \{\text{isNessus2}\} = \text{True} \ \text{ping -w 20 nmap -p}$

$\{\text{NESSUS_PORT}\}$

$\{\text{fail_check}\} = \text{Set Variable If } \{\text{isNessus2}\} = \text{True} \ \text{100% packet loss closed}$

$\{\text{rc}\} \ \{\text{output}\} = \text{Run And Return Rc And Output } \{\text{command}\} \ {\{\text{NESSUS_IP}\}}$

$\{\text{stat}\} = \text{Run Keyword And Return Status Should Not Contain } \{\text{output}\} \ \{\text{fail_check}\}$

#if machine is down before having generated the token it's ok to switch machines, if it is afterwards then the scan can't continue

$\{\text{tryAgain}\} = \text{Run Keyword If } \{\text{beforeGettingToken}\} = \text{True and } \{\text{isNessus2}\} = \text{False} \ \text{Set Variable True}$

Run Keyword If $\{\text{beforeGettingToken}\} = \text{False} \ \text{Should Be True } \{\text{stat}\} \ \text{Nessus Server went down, existing scan now} \\text{\n}$

#If the machine is down and it is nessus_ip2, then it means that both machines are down and the test should \ stop

Run Keyword If $\{\text{stat}\} = \text{False and } \{\text{isNessus2}\} = \text{True} \ \text{Should Be True } \{\text{stat}\} \ \text{Both Nessus Servers available are down} \\text{\n}$

[Return] $\{\text{tryAgain}\}$

generate nessus token

$\{\text{rc}\} \ \{\text{output}\} = \text{Run And Return Rc And Output } \text{curl -s -k --data username=${NESSUS_USER}&password=${NESSUS_PASSWORD}}" \ --no-proxy ${NESSUS_IP} \ https://${NESSUS_IP}${NESSUS_PORT}/session | \text{jq-linux64} '.token' | \text{sed}'s/\^.*\$/1/'$

$\{\text{isNotEmpty}\} = \text{Run Keyword And Return Status Should Not Be Empty } \{\text{output}\}$

Run Keyword If $\{\text{isNotEmpty}\} = \text{True} \ \text{Set Test Variable } \{\text{TOKEN}\} \ \{\text{output}\}$

$\{\text{tryAgain}\} = \text{Run Keyword If } \{\text{isNotEmpty}\} = \text{False} \ \text{check availability and retry True}$

Run Keyword If $\{\text{tryAgain}\} \ \text{set nessus args } \{\text{NESSUS_IP2}\} \ \{\text{NESSUS_PORT2}\} \ \{\text{NESSUS_USER2}\} \ \{\text{NESSUS_PASSWORD2}\}$

[Return] $\{\text{tryAgain}\}$

get scan id

[Arguments] $\{\text{token}\} \ \{\text{policy}\} \ \{\text{scanname}\} \ \{\text{targets}\} \ \{\text{email_recipient}\}$

#list existing policies

$\{\text{rc}\} \ \{\text{response}\} = \text{Run And Return Rc And Output } \text{curl -s -k -H "X-Cookie:}
token=${token}" --noproxy ${NESSUS_IP} https://${NESSUS_IP}${NESSUS_PORT}/policies | jqlinux64 ".policies[] | {name: .name, id: .id, uuid: .template_uuid}"

${isNotEmpty}= Run Keyword And Return Status Should Not Be Empty ${response}
Run Keyword If ${isNotEmpty}==False check availability and retry False

#get policy id for the given scan
${policy_list}= Split String ${response} }
Log ${policy_list}

${policy_match}= Get Matches ${policy_list} **"name":${SPACE}"${policy}"**
${policy_string}= Convert To String ${policy_match}
${aux}= Fetch From Right ${policy_string} "id":${SPACE}
${policyid}= Fetch From Left ${aux} ,
Log ${policyid}

#Get template uuid for the given scan
${aux}= Fetch From Right ${policy_string} "uuid":${SPACE}"
${templateuuid}= Fetch From Left ${aux} 
Log ${templateuuid}

#Get folder id
${folderid}= get folder id

#Get scan id

${rc} ${scanid}= Run And Return Rc And Output curl -s -k -H "X-Cookie: token=${token}"
\n-H 'Content-Type: application/json' -d "{"uuid":"$templateuuid"", ","settings": ",\"name\": ",\"scannename\": ",\"text_targets": ",\"targets\": ",\"emails\": ",\"email_recipient\": ",\"policy_id\": ",\"policyid\": ",\"folder_id\": ",\"folderid\"" -noproxy ${NESSUS_IP} https://${NESSUS_IP}${NESSUS_PORT}/scans | jq-linux64 ".scan.id"

${isNotEmpty}= Run Keyword And Return Status Should Not Be Empty ${scanid}
Run Keyword If ${isNotEmpty}==False check availability and retry False
Set Test Variable ${SCANID} ${scanid}
[Return] ${scanid}

run nessus scan

[Arguments] ${token} ${scanid}
get found vulnerabilities

[Arguments] ${token} ${scanid} ${year} ${month} ${day}

wait for scan completion ${token} ${scanid}

#create file for robot report

${results_summary}= Set Variable
$NESSUS_REPORTS/$lab_name/$test_name/CriticalAndHighVulnerabilities_${year}${month}${day}.txt

Set Test Variable ${results_summary} ${results_summary}

${rc}= Run And Return Rc chmod -R 777 $NESSUS_REPORTS/

Create File ${results_summary} OperatingSystem.Append To File ${results_summary} Short Report on Critical and High Severity Vulnerabilities Found

#now that the scan is complete, the robot can retrieve the vulnerabilities

${rc} ${details}= Run And Return Rc And Output curl -s -k -H "X-Cookie: token=${token}" --noproxy $NESSUS_IP https://$NESSUS_IP$NESSUS_PORT/scans/$scan_id | jq-linux64 ".hosts[] | {host_id: .host_id, critical: .critical, high: .high, hostname: .hostname}"

${isNotEmpty}= Run Keyword And Return Status Should Not Be Empty ${details}

Run Keyword If ${isNotEmpty}==False check availability and retry False

@{details_list}= Split String ${details} }

${plugins_hosts}= Create Dictionary

: FOR ${host_details} IN @{details_list}

\ ${empty}= Run Keyword And Return Status Should Not Contain ${host_details} host_id

\ Exit For Loop If ${empty} #for some reason I cannot remove the last newline in the list, but it works like this as well

\ ${clean_details}= Remove String ${host_details} \n 

\ ... ${SPACE}
\[$\{aux\} = \text{Fetch From Left} \ \$\{clean\_details\} \ \text{critical}\]
\[$\{hostid\} = \text{Fetch From Right} \ \$\{aux\} \ \text{host\_id:}\]
\[$\{aux\} = \text{Fetch From Left} \ \$\{clean\_details\} \ \text{high}\]
\[$\{critical\_vulns\} = \text{Fetch From Right} \ \$\{aux\} \ \text{critical:}\]
\[$\{aux\} = \text{Fetch From Left} \ \$\{clean\_details\} \ \text{hostname:}\]
\[$\{high\_vulns\} = \text{Fetch From Right} \ \$\{aux\} \ \text{high:}\]
\[$\{hostIP\} = \text{Fetch From Right} \ \$\{clean\_details\} \ \text{hostname:}\]
\[$\{hostIP\} = \text{Remove String} \ \$\{hostIP\} \ \$\{SPACE\} \ \n\]
\[$\{critical\_or\_high\} = \text{Run Keyword And Return Status} \ \text{Should Be True}\]
\[$\{critical\_vulns\} == 0 \text{ and } \{high\_vulns\} == 0\]
\[$\{vuln\_found\} = \text{Set Variable If} \ \$\{critical\_or\_high\} == \text{False} \ True \ True \ False\]
\[\text{Run Keyword If} \ \$\{critical\_or\_high\} == \text{False} \ \text{sum up results} \ \$\{TOKEN\} \ \$\{scanid\} \ \$\{hostid\}\]
\[\ ... \ \$\{hostIP\} \ \$\{plugins\_hosts\}\]
\[@\{plugins\_split\} = \text{Get Dictionary Keys} \ \$\{plugins\_hosts\}\]
\[$\{results\} = \text{OperatingSystem.Get File} \ \$\{results\_summary\}\]
\[: \text{FOR} \ \$\{plugin\} \ \text{IN} \ @\{plugins\_split\}\]
\[$\{hosts\} = \text{Get From Dictionary} \ \$\{plugins\_hosts\} \ \$\{plugin\}\]
\[\text{Set Test Variable} \ \$\{\{plugin\}_plug\} \ \$\{hosts\}\]
\[$\{results\} = \text{Replace Variables} \ \$\{results\}\]
\[$\{final\} = \text{Convert To String} \ \$\{results\}\]
\[\text{Remove String} \ \$\{final\} \ \text{False} \ \text{\n} \ True \ \text{\n} \ \text{#False coming from run keyword if output is unwanted}\]
\[$\{final\} = \text{Encode String To Bytes} \ \$\{final\} \ \text{UTF-8}\]
\[\text{OperatingSystem.Create File} \ \$\{results\_summary\} \ \$\{final\}\]
\[\text{OperatingSystem.Copy File} \ \$\{results\_summary\}\]
\[$\{SC\_HOME\}/\text{reports/NessusReports/}$\{test\_name\}/\]
\[\[\text{Return}\] \ \$\{vuln\_found\}\]

\textbf{sum up results}

\[\text{[Arguments]} \ \$\{token\} \ \$\{scanid\} \ \$\{hostid\} \ \$\{hostIP\} \ \$\{plugins\_hosts\}\]
#get vulnerability details

${rc}    ${vulnerability_details}=    Run And Return Rc And Output    curl -s -k -H "X-Cookie: token=${token}" --no-proxy $NESSUS_IP https://$NESSUS_IP$NESSUS_PORT/scans/$scan_id/hosts/$hostid | jq -l
".vulnerabilities[] | {severity: .severity, plugin_name: .plugin_name, plugin_id: .plugin_id} | select(.severity | contains(3) or contains(4))"

${isNotEmpty}=    Run Keyword And Return Status    Should Not Be Empty

${vulnerability_details}

Run Keyword If    ${isNotEmpty}==False    check availability and retry    False

${vulnerability_details}=    Remove String    ${vulnerability_details}    {    ,    "

[@{vuln_split}=    Split String    ${vulnerability_details}    }
 : FOR    ${vuln}    IN    @{vuln_split}

    ${empty}=    Run Keyword And Return Status    Should Not Contain    ${vuln}    severity

    Exit For Loop If    ${empty}    #for some reason I cannot remove the last newline in the vuln_split list, but it works like this as well

    ${severity}=    Fetch From Left    ${vuln}    ${SPACE}plugin_name

    ${severity}=    Fetch From Right    ${severity}    severity:

    ${severity_name}=    Set Variable If    ${severity}==${SPACE}4    Critical    High

    ${plugin_name}=    Fetch From Right    ${vuln}    plugin_name:

    ${plugin_name}=    Fetch From Left    ${plugin_name}    ${SPACE}plugin_id

    ${plugin_id}=    Fetch From Right    ${vuln}    plugin_id:

    ${plugin_id}=    Remove String    ${plugin_id}    ${SPACE}

    ${accepted}=    check if in blacklist    ${plugin_id}

    ${blacklist_info}=    Set Variable If    ${accepted}==True    [Accepted]    [Rejected]

    ${already_reported}=    Run Keyword And Return Status    Dictionary Should Contain Key    ${plugins_hosts}    ${plugin_id}

    #if the plugin wasn't added yet, add it with the respective hostIP

    Run Keyword If    ${already_reported}==False    Set To Dictionary    ${plugins_hosts}    ${plugin_id}    ${hostIP}

    #if the plugin was added, check if the host was added to the plugin

    ${plugin_newHost}=    Run Keyword If    ${already_reported}==True    Get Count    ${plugins_hosts}    ${hostIP}

    Run Keyword If    ${plugin_newHost}==0    add host to plugin    ${plugins_hosts}    ${hostIP}    ${plugin_id}
\ #get plugin output details

\ $[rc]  \${plugin_out}= Run Keyword If \${already_reported}==False  Run And Return

Rc And Output  curl -s -k -H "X-Cookie: token=${token}" --no-proxy ${NESSUS_IP}
https://${NESSUS_IP}$NESSUS_PORT/scans/${scan_id}/hosts/${hostid}/plugins/${plugin_id}
| jq -linux64 ".outputs[] | {plugin_output: .plugin_output}"

\ ${isNotEmpty}= Run Keyword And Return Status  Should Not Be Empty  \${plugin_out}

\ Run Keyword If \${isNotEmpty}==False  check availability and retry  False

\ ${output}= Run Keyword If \${already_reported}==False  parse plugin_out
${plugin_out}

\ ${NAME}= Set Variable \${${plugin_id}_plug}

\ Run Keyword If \${already_reported}==False  OperatingSystem.Append To File
\ ${results_summary}  ${blacklist_info} ${severity_name} vulnerability:${plugin_name} found
on ${NAME}
${output}
Appendix B

Confluence Page Example

This appendix aims to present an example of the confluence pages created with the goal of helping with the future maintenance of the developed tests.

B.1 Main Confluence Page Example

The main confluence page briefly explains what was developed and some of the details regarding the software used. Then, it presents the test suites developed, their respective test cases and how to update the tests, namely some of the variables used for validation as well as the blacklist files.
Security Test Automation

- **Automation Achievement - What?**

A set of security tests was developed under the Robot framework in order to facilitate the detection of vulnerabilities in an earlier development stage.

The security cycle is running on [SFA TEX Proxy 06](https://10.41.96.26) with the following security tools installed:

- **Nmap** (/opt/nmap-7.12.1.x86_64.rpm) which needs
  - Ncat (/opt/ncaat-7.12.1.x86_64.rpm)
  - Nping (/opt/nping-0.7.12.1.x86_64.rpm)
- **Arachni** (/opt/arachni-1.4.0.5.10)

Apart from these tools, it was also installed:

- jdk1.8.0_91 (/opt/jdk1.8.0_91)
- jq-linux64 (/opt/jq-linux64)

The testcases using Nessus are using Nessus server 10.91.215.254:8834 by default and, in case this one isn’t available, Nessus server 10.9.212.124 is chosen.

- **Automation Achievement - How?**

The security cycle is run once a day and performs the following scans:

- **Web Vulnerability Tests - using Arachni**
  - SQL injection test
  - NoSQL injection test
  - Code injection test
  - File inclusion test
  - Cross Site Scripting (XSS) test
- **Port Scanning Tests - using Nmap**
  - TCP scan
  - IP Protocol scan
  - UDP scan
- **Vulnerability Tests - using Nessus**
  - Authenticated local scan
  - Unauthenticated local scan

- **Updating test details**

[SQM Security Pipe UpdatingInfo.pdf]
B.2 Web Vulnerability Tests Confluence Page Example

For each of the test suites, a child page was created, explaining the general structure of the test and a few of its implementation details. In this case, we chose to present the Web Vulnerability Tests confluence page.
Web Vulnerability Tests - using Arachni

Apart from the web user interface, Arachni also has a command line interface available. These robots were implemented taking advantage, mostly, of the command line interface.

- Each robot starts by getting the target IP (SDV_CPF_LB_WAS_ACCESS_ADDRESS) dynamically from the sdv.py file
- From there, arachni is called as follows: arachni --profile-load-filepath ${testing_profile_path} https://${target_ip} --browser-cluster-pool-size 6 --timeout 2:45:0 --report-save-path=${ARACHNI_REPORTS}
  - The only thing that varies from test case to test case (inside the web vulnerability tests test suite) is this testing profile so - as there are five test cases - there are five testing profiles, each with the following security checks:
    - SQL injection
      - 3 active checks (which actively engage the web application via its inputs): SQL injection, Blind SQL injection (differential analysis), Blind SQL injection (timing attack)
      - 1 passive check (which will passively collect data): Interesting responses
    - XSS
      - 8 active checks: XSS, DOM XSS, DOM XSS in script context, XSS in HTML element event attribute, XSS in path, XSS in script context, XSS in HTML tag, CSRF
      - 1 passive check: Interesting responses
    - NoSQL injection
      - 2 active checks: NoSQL injection, Blind NoSQL Injection
      - 1 passive check: Interesting responses
    - Code injection
      - 3 active checks: Code injection, Code injection (php://input wrapper), Code injection (timing)
      - 1 passive check: Interesting responses
    - File inclusion
      - 2 active checks: File inclusion, Remote File inclusion
      - 6 passive checks: Interesting responses, Backdoors, Backup directories, Backup files, Common directories, Common files
      - NOTE: the testing profiles were created using the Arachni web interface for simplicity reasons.
  - Apart from the security checks, every created profile has the following plugins enabled
    - AutoThrottle - Monitors HTTP request times and automatically throttles the request concurrency in order to maintain stability and avoid from killing the server
    - Discovery check response anomalies - Analyzes the scan results and identifies issues logged by discovery checks (i.e. checks that look for certain files and folders on the server), while the server responses were exhibiting an anomalous factor of similarity
    - Health map - Generates a simple list of safe/unsafe URLs
    - Metrics - Captures metrics about multiple aspects of the scan and the web application
    - Timing attack anomalies - Analyzes the scan results and logs issues that used timing attacks while the affected web pages demonstrated an unusually high response time; a situation which renders the logged issues inconclusive or (possibly) false positives
    - Uniformity (Lack of central sanitization) - Analyzes the scan results and logs issues which persist across different pages. This is usually a sign for a lack of a central/single point of input sanitization, a bad coding practise
  - After the scan is over it produces a .txt file which works as a "pre-report". From that file and using the arachni-reporter executable the robot builds two reports:
    - arachni-reporter "$scan_output" --reporter=/html:outfile="$report_pathHTML" : an html report which is saved to be verified by the human tester later
    - arachni-reporter "$scan_output" --reporter=/json:outfile="$report_pathJSON" : a json report which is used by the robot to decide whether the test passes or fails
  - As in the Vulnerability Tests, jq-linux64 to help with parsing the arachni .json reports during the validation step. This step is performed considering not only the vulnerabilities found but also a blacklist file which should be updated each time there is a new SQMPRODUCT release
    - every time a vulnerability is found the robot checks if it is listed in the blacklist file. If it is contained, it means that the vulnerability is already known to exist and it is inserted in the report marked as [Accepted]. On the other hand, if it isn't contained the vulnerability is marked as [Rejected]
    - if there is one or more vulnerabilities marked as [Rejected] in the report the test fails
    - Note: the only vulnerabilities considered are the ones with High and Medium severity (there is no critical severity in Arachni).
- Finally, the structure of the short report generated is:
  - Short Report on High and Medium Severity Vulnerabilities Found (Report Title)
  - Blacklist updated for SQMPRODUCT <version in the blacklist file> and the current version <version in the lab ViiS> (note to point out when the blacklist file should be updated)
  - <Blacklist mark>
  - vulnerability_name: <vulnerability name>
    - cwe: <cwe>
    - vulnerability_severity: <severity> (the only two possible values are High or Medium)
    - vulnerability_solution: <vulnerability solution suggested by Arachni>
    - checkName: <active check name that found the vulnerability>
    - checkDescription: <check description>

• **vectorClass**: `<class of the attack vector>`
• **vectorUrl**: `<vector url target>`
• **vectorSource**: `<vector source code>`
• **vectorMethod**: `<vector method>`

**Example:**

When validation is over, as well as all the test cases in the test suite, all the .json and .afr reports are erased and the test terminates.