Security Testing in Continuous Integration Systems

Mariana Cristina Figueiredo Paulo
mariana.f.paulo@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2016

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.

1. Introduction

Software testing is an important part of a software's lifecycle and, among the several existing *software testing* categories, there is security testing.

The main goal of software security is to assure a proper software behavior if the system suffers a malicious attack [7]. 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.

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. 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.

Nokia wants to make its products stand out from the security point of view. 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.

2. Related Work

Each of the several existing security testing methodologies has its advantages and disadvantages, much related to the type of vulnerabilities that it is and is not able to detect and with the effort required to do so. In this section, some examples of these methodologies are introduced, as well as the continuous integration concept.

[1] `http://networks.nokia.com/portfolio/products/operations-support-systems/service-quality-manager`

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. Its main advantage is that the tester does not need to know the specifics of a program. Still, by not considering the program's inner details, a lot of vulnerabilities may not even be considered.

An example of this approach is vulnerability scanning, which stands for using an automated tool to analyze web applications to verify the existence of vulnerabilities without access to the application's source code [1].

In the work presented by Bau *et al.* in [1], assessing the current state of the art of automated black-box web application vulnerability testing, it was possible to conclude that these scanners mainly devote their attention to information leakage vulnerabilities, followed by XSS and SQL injection vulnerabilities.

2.2. White and Grey-box Security Testing

There are several types of vulnerabilities that the black-box approach does not identify, since they are not even considered. Grey and white-box security testing solve this problem by concentrating on those disregarded vulnerabilities.

When using a white-box technique, 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.

Static Code Analysis is an example of a white-box approach, leveraging the program's code to identify possible vulnerabilities. This identification is based on already known risks associated with the use of certain coding practices [5]. Between its many applicances, Wasserman and Su presented a static analysis tool for finding XSS vulnerabilities addressing the problem of weak or even absent input validation [11].

Check 'n' Crash (CnC) is a grey-box security tester which combines static checking and concrete test-case generation [2]. It is a combination of two tools: ESC/Java is the static checking system that analyzes a program, reasons abstractly about unknown variables and tries to detect errors, generating the respective reports.; JCrasher, taking the generated error reports and produces Java test cases that activate the existing errors.

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. Among other things, it helps with understanding what the attack goals are, who the attackers are and which attacks are likely to occur [8].

The SDL (Security Development Lifecycle) threat modeling process, according to Microsoft, can be synthesized into four main steps: diagramming; threat identification; mitigation and validation [6] [10]. The diagramming phase helps with describing the boundaries of the system and can provide a good level of detail of the system components and information flow. Regarding the threat identification phase, it focuses on identifying threats for each of the diagram components, making use of the STRIDE vulnerabilities taxonomy. The third phase represents the main goal of doing threat modeling, mitigate the identified threats and the fourth phase stands for validating the threat model.

Following Microsoft's threat modeling approach, Marback *et al.* presented a system intending to automatically generate security tests from threat trees [4]. To do so it analyzes threat trees created by the Microsoft Threat Modeling Tool[2], producing test sequences which are then transformed into executable tests. The experimental results of this tool's usage demonstrated that it is effective towards discovering unmitigated threats presenting few false positives or negatives.

2.4. Continuous Integration

Continuous integration (CI) is a software development practice, mostly related to how code is revised, tested and integrated into the project [3]. In this process developers frequently commit their source code changes or additions to a repository, triggering a build which is followed by tests [9]. 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 reported bugs.

There is a multitude of CI software frameworks that can be used to automate both the build and the testing steps. Seth and Khare proposed a continuous integration and test automation implementation [9]. Their approach was to implement Jenkins[3] in a master slave architecture for daily integration of continuous build and testing. The obtained results after this implementation showed that the employee work-hour efficiency was enhanced.

3. SQM Security Pipe

The present section will explain how the work was developed, going into the environment in which this work is inserted in, the security testing tools used and the steps taken towards planning and implementing the tests.

3.1. Environment

In order to briefly describe this work's environment we start by introducing the product to which the

[2]`https://www.microsoft.com/en-us/download/details.aspx?id=42518`

[3]`https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins`

tests will apply to (SQM) and, then, evolve into presenting the automation tool used to develop the 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. 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.

Currently, SQM is based in a virtualized environment. Its architecture 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. Being used for test automation means that the tests run with minimal or even without any level of manual actions being performed.

This is an easy to use Framework through the RIDE editor support[4] and can provide clear reports and detailed logs. These 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[5].

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 because, as it was mentioned before, 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:

[4]`https://github.com/robotframework/RIDE/wiki`

[5]`http://www.slideshare.net/pekkaklarck/robot-framework-introduction`

1. A developer makes a commit to project X into an SVN repository[6];

2. Jenkins[7] 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.

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. 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.

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 and, from that, build a Level 1 and a Level 2 Data-Flow diagram. From those, we were able to see the main components, their sub-components and the information flow between them. Making it easier to understand how the information is processed and sent and how this data trades can affect the entire system.

Using the DFDs it is, then, possible to move to the SDL's second phase: threat identification. Table 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:

[6]`http://subversion.apache.org/`

[7]`https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins`

Element	S	T	R	I	D	E
NetAct Data Collector	X	X	X	X		
Event Collection Framework		X	X	X	X	
Event Engine		X	X	X	X	
DB Oracle	X	X		X	X	X
JMS Queue Provider	X	X	X	X		
Service Queue Worker	X	X	X	X		
Rule Engine Servlet		X				
Engine Factory						
Root Cause Analysis Servlet		X				
Service Alarm Manager		X		X	X	X
SQM GUI	X			X	X	X

Table 1: Threat Identification of the SQM components

- 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.

SDL's 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 describe these tools, the features that were used and, also, the reasoning about these choices.

3.3.1. Arachni - Web Vulnerability Scanner

Arachni[8] 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.

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[9] 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 available, we used the *Advanced Scan* type which 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,

[8]http://www.arachni-scanner.com/

[9]https://www.tenable.com/products/nessus-vulnerability-scanner

namely its name, policy and target IP(s).

Regarding the policy, there is also a variety of preconfigured options that the user can choose from to best fit its needs. 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[10] 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 and IP Protocol scan, which will be described in Section 3.4.

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.

3.4. Test Development

As a part of a Nokia process for developing secure products and systems, SQM 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. 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. That build is then 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.

[10]https://nmap.org

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 an *SQL Injection* robot, an *XSS* robot, a *NoSQL Injection*, a *Code Injection* robot and a *File Inclusion* robot. Each of these has its own active and passive checks.

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. In this case, for the test suite, we created a keyword named *prepare lab dir* which manages the existence of a directory which will store the test suite results, as well as its permissions. After 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*, managing the test case directories for each test suite.

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, being 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 ($SDV_CPF_LB_WAS_ACCESS_ADDRESS) 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 an arachni plugin which, from the .afr report, 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[11] 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 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 is a .txt file which indicates the last product version to which it was updated and a list of the desired vulnerabilities. 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.

[11]https://stedolan.github.io/jq/

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.

The test cases start with the keyword (*define scan targets*) which dynamically defines the targets of the vulnerability scan. 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.

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. If a Nessus token is successfully received the robot moves to the *get scan id* keyword, creating 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. 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* keyword is used to create the scan targets list.

With all the targets defined, the core of the test starts, being based on a *for* cycle. This cycle goes through each node and performs the following actions:

- Nmap is executed 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;

- The *parse ports info* keyword gets the report and parses its information, obtaining a list of pairings that 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. With that list, the robot uses the *service Port Dictionary Creation* keyword to parse it and build a a list of pairings 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, 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 or an unexpected service is found 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.

Test Suite	**Test Cases**	**Main Structure**	**Keywords**	**Test Case (Total)**
Web Vulnerability	SQL Injection	10 (lines)	10 (per test case)	104 (lines)
	XSS			
	NoSQL Injection			
	Code Injection			
	File Inclusion			
Vulnerability	Authenticated Scan	19 (lines)	19 (per test case)	285 (lines)
	Unauthenticated Scan			
Port Scanning	TCP Scan	37 (lines)	10 (per test case)	184 (lines)
	UDP Scan			
	IP Protocol Scan			

Table 2: Developed Test Suites and Test Cases Summary

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 2 for the web vulnerability test suite, each of 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, 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.

For future maintenance of this work, a Confluence[12] – a team collaboration wiki software – page was created explaining the developed work and some of the details regarding the setup of the environment where it is inserted in.

4. Evaluation

In this section we will explain why the previously existing solution could not be used anymore, how the developed work improved the previous solution

[12]`https://www.atlassian.com/software/confluence`

and, also, which are the advantages it brings to SQM's development.

It is important to denote that we felt the need to present 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. 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 tools and tests configurations;

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, resulting in unsolved compatibility problems. 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 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 constantly updating those details on the files 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 Section 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* - <u>achieved a medium level of automation with Nmap and Nessus scripts.</u> 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* - <u>designed 5 types of security scans: port scanning, vulnerability scanning, robustness testing, DoS testing and web vulnerability scanning.</u> 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* - <u>achieved test automation for vulnerability scanning, port scanning, web vulnerability scanning and robustness testing</u>. 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. In addition, result analysis was left to be performed manually, which 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:

- 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, shown in Table 3, which demonstrate the gains that SQM Security Pipe brings, when compared to what existed before the STAS solution and with the STAS solution. 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.

Metrics	**Before STAS**	**With STAS**	**With SQM Security Pipe**
Security Cycle Run Time	120 (hours)	36 (hours)	9 (hours)
Security Cycles Run	4 (per year)	52 (per year)	365 (per year)
Results Analysis Effort	2 (days)	2 (days)	1 (day)
Feedback Time	7 (days)	2 (days)	1 (day)
Opened Prontos	unknown	3	0

Table 3: Security Cycle Metrics - before the STAS solution, with the STAS solution, with the SQM Security Pipe solution

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*[13]. This is an

[13]Pronto is a Nokia's tool for managing faults found in

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.

In order to justify not only the aim but also need for this work, Engineer José Ramos started by introducing the Nokia Design For Security process 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 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.

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

5. Conclusions

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 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.

Nevertheless, the tests were designed and developed 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. Adding to this, we were able to successfully automate and continuously integrate the developed tests. So, all in all, the main goal of designing, developing and integrating a set of security tests into Nokia's build pipe was fulfilled.

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 developed. 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.

References

[1] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[2] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE '05 Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, 2005.

[3] S.-T. Lai and F.-Y. Leu. Applying Continuous Integration for Reducing Web Applications Development Risks. *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, pages 386–391, 2015.

[4] A. Marback, S. Kondamarri, and D. Xu. Security test generation using threat trees. *ICSE '09 Workshop on Automation of Software Test*, pages 62–69, 2009.

[5] I. Medeiros, N. F. Neves, and M. Correia. Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives. In *WWW'14 Proceedings of the 23rd International Conference on World Wide Web*, pages 63–74, 2014.

[6] Microsoft. Introduction to Threat Modeling, 2011.

[7] B. Potter and G. McGraw. Software security testing. *IEEE Security & Privacy Magazine*, 2(5):81–85, 2004.

[8] B. Schneier. Attack Trees. *Dr Dobbs Journal*, 24(12):21–29, 1999.

[9] N. Seth and R. Khare. ACI (Automated Continuous Integration) using Jenkins : Key for Successful Embedded Software Development. *Proceedings of 2015 RAECS UIET Panjab University Chandigarh 21-22nd December 2015*, (December), 2015.

[10] A. Shostack. Experiences threat modeling at Microsoft. In *MODSEC '08 Proceedings of the Workshop on Modeling Security*, volume 413, 2008.

[11] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08 Proceedings of the 30th international conference on Software engineering*, pages 171–180, 2008.