Universitat Oberta de Catalunya

uoc.edu

Application Security Testing Tools Study and Proposal

Miro Casanova Páez
Máster Universitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones. (MISTIC)
Seguridad Empresarial

Pau del Canto Rodrigo
Victor Garcia Font

12/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	Application Security Testing Tools Study and Proposal
Nombre del autor:	Miro Casanova Páez
Nombre del consultor/a:	Pau del Canto Rodrigo
Nombre del PRA:	Victor Garcia Font
Fecha de entrega:	12/2020
Titulación:	Máster Universitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones (MISTIC)
Área del Trabajo Final:	Seguridad Empresarial
Idioma del trabajo:	Inglés
Palabras clave	SAST, DAST

Resumen del Trabajo

Hoy en día es una evidencia la necesidad de un menor tiempo de lanzamiento al mercado de las aplicaciones. Sin embargo, aunque el tiempo necesario para desarrollarlas se reduzca, todavía se necesita poder ofrecer aplicaciones confiables y seguras. Esto era ya es una tarea difícil, y ahora lo es aún más con las restricciones de tiempo y la rápida evolución de las tecnologías. Los errores ocultos en el software pueden traducirse en vulnerabilidades de seguridad que potencialmente permitan a los atacantes poner en peligro los sistemas y aplicaciones.
Hay hackers y crackers que pueden estar al acecho para obtener nuestra valiosa información personal. Por lo tanto, estas aplicaciones deben ser seguras y confiables ya que nuestra información o documentos privados se almacenan en estas aplicaciones de n capas. Cada año miles de estas vulnerabilidades se desvelan públicamente en el *Common Vulnerabilities and Exposures database* [23]. Estas vulnerabilidades son a menudo ocasionadas por errores sutiles cometidos por programadores y pueden propagarse rápidamente debido a la prevalencia del software de código abierto así como a la reutilización de código.
Nos enfrentamos al dilema de la necesidad de acelerar el proceso de desarrollo de software y al mismo tiempo el requisito de ofrecer aplicaciones fiables y seguras. Hay muchos enfoques para abordar este problema que van desde adaptar el proceso de desarrollo de software a soluciones técnicas más concretas. En este TFM intentaremos analizar el uso de una o varias herramientas automatizadas para verificar si las aplicaciones en construcción tienen el nivel de seguridad requerido mediante la detección de posibles vulnerabilidades o errores que podría causar un funcionamiento no deseado. Este enfoque aborda la detección de código vulnerable durante el curso del ciclo de desarrollo de software.

Abstract:

Nowadays the need for a shorter time-to-market of applications is evident. However, even though the time needed for developing them gets reduced, we still need to be able to deliver reliable and secure apps. This was already a challenging task, and it is even more so with the time restrictions and the rapidly evolving technologies. Hidden flaws in software can result in security vulnerabilities that potentially allow attackers to compromise systems and applications.
There are hackers and crackers who may be keeping an eye on our valuable personal information. Hence, these applications need to be secured and should be reliable since our private and important information or documents are stored on the back end of these n-tiered applications. Each year thousands of such vulnerabilities are reported publicly to the Common Vulnerabilities and Exposures database [23]. These vulnerabilities are often caused by subtle errors made by programmers and can propagate quickly due to the prevalence of open-source software and code reuse.
We are confronted with the dilemma of the need for speeding up the software development process while at the same time the requirement of delivering reliable and secure applications. There are many approaches for tackling this problem which range from adapting the software development process to more concrete technical solutions. In this TFM we will try to analyse the use of one or several automatized software tools for verifying whether the application under construction has the required level of security by detecting potential vulnerabilities or flaws that could cause an undesired misfunction. This approach addresses the detection of vulnerable code during the course of the software development cycle.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and motivation

Nowadays, the need for a shorter time-to-market of applications is evident. However, even though the time needed for developing them gets reduced, we still need to be able to deliver reliable and secure apps. This was already a challenging task, and it is even more so with the time restrictions and the rapidly evolving technologies. Hidden flaws in software can result in security vulnerabilities that potentially allow attackers to compromise systems and applications. Some hackers and crackers may be keeping an eye on our valuable personal information. Hence, these applications need to be secured and should be reliable since our private and important information or documents are stored on the back end of these n-tiered applications. Each year thousands of such vulnerabilities are reported publicly to the Common Vulnerabilities and Exposures database [23]. These vulnerabilities are often caused by subtle errors made by programmers and can propagate quickly due to the prevalence of open-source software and code reuse.

We are confronted with the dilemma of the need for speeding up the software development process while at the same time the requirement of delivering reliable and secure applications. There are many approaches for tackling this problem which range from adapting the software development process to more concrete technical solutions. In this TFM we will try to analyse the use of one or several automatised software tools for verifying whether the application under construction has the required level of security by detecting potential vulnerabilities or flaws that could cause an undesired misfunction. This approach addresses the detection of vulnerable code during the course of the software development cycle.

1.2 Objectives

The first objective of this TFM is to propose a suitable solution for using one or many of the tools, reviewed in this document, to be incorporated in the software development cycle for improving the delivery of more reliable and secure applications, according to a set of criteria to be defined below. These

criteria are based on the current situation of the organisation where we currently work.

To be able to attain the previous objective, we need to perform a profound literature study of the different tools (and kinds of tools) available in the market that can be applicable to solve the aforementioned problem. We will also search for benchmarks and comparisons of these tools to analyse their advantages and disadvantages with respect to each other.

We will compare the different kinds of tools and evaluate them with respect to a set of criteria. A summary of those criteria are (amongst others):

- The use of the tool must not interfere with the normal development of the application.
- The tools should be ideally automatised (or mostly automatised) without the need for the intervention of developers.
- The tools must be easy to use.
- The tools should be effective, i.e. they should have a low rate of detection of false positives (potential problems detected by the tools that actually are not) and a high rate of detection of true positives (potential problems that are actually real vulnerabilities).
- The output of these tools must be easy to understand by developers so that they can take action immediately to remediate the situation
- The tools must support a certain set of technologies in which the applications are developed, being Java (Spring), TypeScript (Angular), and Atlassian tools the most important.
- The cost of the tools must be reduced. Ideally, these should be free. However, if there is a non-free tool much better than an open source according to these criteria, its recommendation should also be pondered.

The final objective is to define a set of tools, for the aforementioned organisation, with the set of criteria described above as well as some technological restrictions, explained in detail in chapter 4. We will also try to immediately integrate some tools in case we see it is a quick win, i.e. tools that do not need a lot of configuration and that can be installed locally on the developers' workstation (and thus not on the servers).

We would like to clarify that any risk analysis is out of the scope of this TFM. Performing a rigorous cost/benefit (such as return on investment ROI) of implementing the tools in the organisation is out of the scope of this work as well. It would take longer to perform those two activities than what we have available for this TFM, and it would be necessary to have access to both demo or trial versions of commercial tools as well as their licensing conditions.

1.3 Methodology

First, we will perform an extended literature study of all the kinds of tools and the tools that may be applicable to attain our objectives. By extended we mean that we will not only analyse the characteristics of each (kind of) tool, including their benefits and drawbacks, but we will also look at the existing comparisons and benchmarks that have been performed so far and their corresponding conclusions. This will be the one of our contributions.

Our initial idea is that we are not going to perform benchmarks of the tools ourselves. We will try to find them in the existing literature. However, if we see that some interesting tool (according to its specifications) appears, that excels its competitors by far and has not been reviewed in the literature, we will do some benchmarking in case the licensing of the product allows it. The criteria for the realisation of the benchmark will be defined if needed. In case we do the benchmark, it will be another contribution of our work.

Afterwards, we will analyse the convenience of some of the tools according to the criteria described above so that they can be integrated by the organisation. We will accordingly choose the set of tools to be used to improve the development process and in this way achieve our objectives. As already said above, we will study the feasibility of integrating some tools in the current software development ecosystem of the organisation.

1.4 Planning

- Review the state of the art of the kinds of tools suitable for this TFM. 2 weeks (13th October).
- Detect for each kind the tools that might be useful according to our criteria (and some other potential criteria that we might consider relevant). 2 weeks (27th October).
- Review literature and search for benchmarks and comparison of those tools. 2 weeks (10th November)
- Eventually perform benchmarks for some of the tools in case the tools seem interesting, no unbiased benchmark of them has been found, and the licensing allows us. 3 weeks (1st December).
- Based on the previous, we will conclude which tools are the most suitable for our case and argue how they can be integrated in the software development cycle. 2 weeks (15th December).
- Corrections based on feedback. 2 weeks (29th December).
- Preparation of the video (5th January).

- Defense of TFM (15th January).
- Writing the TFM will be an ongoing task until 29th December.

1.5 Summary of the obtained product

In this TFM there are essentially two products. The first, and in our opinion the most important contribution, is the extensive literature study of the different kinds of software security assurance tools, the different tools in the market, and their characteristics. We have tried to avoid going to look for information on the vendors' websites, since they always claim that their products are the best of all. We have taken enough time to search for independent sources of information, especially scientific published articles. The state of the art is divided in chapters 2 and 3.

Our second contribution is the selection of the tools that might be useful for our organisation based on some selection criteria. This is explained in chapter 4. Furthermore, we integrate some of the tools and show some of the results.

1.6 Description next chapters of this work

In chapter 2 we discuss in detail the different kinds of software security assurance tools, namely static application security testing (SAST), dynamic application security testing (DAST), interactive application security testing (IAST) and software composition analysis (SCA) tools. For each of these types of tools, we look at their characteristics, advantages and shortcomings. Also, we discuss some of the specific tools that can be found on the market.

In chapter 3 we discuss in detail the different approaches taken to perform benchmarks of the software security assurance tools. There are some public approaches, such as the OWASP Benchmark project or WAVSEP, that have been used for evaluating the performance of the tools. Also, we review some benchmarks that have been proposed in the academic world.

In chapter 4, we present and discuss the selection criteria for the choice of the software security assurance tools. After, we present the technological ecosystem of the organisation where the tools will be implemented. Based on the criteria and the ecosystem, we make the final selection of the tools to be proposed. Finally, we show the implementation of some of the tools that have been considered to be quick wins, due to their small amount of configuration and small impact on the development of applications. We also show some of the analyses run by those tools on existing projects of the organisation.

Finally, in chapter 5, we conclude.

Chapter 2

State of the art

Software security assurance tools are pieces of software that help detecting or preventing security weaknesses. These weaknesses may result in vulnerabilities, i.e. a possible way of harming the system [14].

There are many ways to look at a piece of software to try to improve it and there are different categories of software analysis tools that can be established according to various criteria. One of the most relevant and interesting criteria is whether a process is manual or automatic. Manual software improvement processes involve a developer or a quality analysis expert to actively evaluate the piece of software, make use of their previously acquired knowledge to understand it and identify its weaknesses. Automatic methods consist of a tool that performs a predetermined series of tests on the piece of software it is given and produces a report of its analysis.

Another criterion commonly used in differentiating the various existing methods of analysing software is the approach they take. A method can either use a black-box approach, considering the whole piece of software as an atomic entity, or a white-box approach, examining the inner workings of the software, e.g. its source code. The black-box approaches most often consist of trying to give the piece of software being analysed unexpected input parameters or unexpected values, thus hoping to trigger incorrect behaviour. The white-box approaches vary significantly in their implementations and can for instance simply look for certain patterns in the code, reconstruct the logic or data flow of the software or even attempt to simulate all possible executions of the program.

You can also look at which stage of the software development life cycle these tools can be used to identify vulnerabilities and weaknesses. Some tools perform the evaluation early in the development cycle, namely during the implementation phase, and produce reports of parts of code that might be vulnerable according to a set of rules or policies. This gives immediate feedback to developers of potential issues, and therefore they can analyse the report and correct the code if necessary. Some other tools can perform the evaluation a bit later in the development cycle, while the applications are already in a running state. This can happen late when the application is ready to go to production as a penetration test, or earlier while the application is not

yet ready to be released but it can be deployed and run.

Another criterion, somehow orthogonal to the previous ones, is to look at which part of the applications are analysed by the tools. Some of them perform the tests of the proprietary code, while other tools analyse the dependencies (third-party libraries) used by the proprietary code being developed. Due to the extensive use of third-party libraries in the development of applications, especially open source, this criterion becomes important.

2.1 SAST, DAST, IAST, SCA Overview

The most common way of classifying the software assurance tools is according to whether they have access to the source code (white box) or they look at the applications as a whole (black box).

The tools that are used to analyse the proprietary source code (or bytecode) of the application to find potential security weaknesses while the code is still in a static/non-running state are called static application security testing (SAST) tools.

A second kind of tools is called web application vulnerability scanners or dynamic application security testing (DAST) tools. These tools scan web applications from the outside and perform penetration testing, i.e. a passive and active analysis of a web application in operating state by simulating attacks on it. Therefore, they do not look at the source code of the application.

A third kind of tools is called interactive application security testing (IAST) tools. IAST tools analyse code for security vulnerabilities while the app is run by an automated test, human tester, or any activity interacting with the application functionality [51]. This technology reports vulnerabilities in real time, which means it does not add any extra time to your continuous integration/continuous delivery (CI/CD) pipeline.

A fourth kind of tools refers to the criterion of looking at the dependencies (third-party libraries) of an application. These tools are called software composition analysis (SCA) tools. They detect and track all open source components in an organisation's code base, to help developers manage their open source components. Due to the increasing importance and massive use of open-source components and libraries, SCA tools are becoming essential for software security assurance.

Beyond these traditional tools, there has been significant recent work on the usage of machine learning for program analysis. The availability of large amounts of open-source code opens the opportunity to learn the patterns of software vulnerabilities directly from mined data [33]. We are not going to explore in detail this innovative kind of tools in this TFM.

2.2 SAST

As we said above, SAST tools are designed to analyse source code or compiled versions of code to help find security flaws when the code is in a non-running state.

There are several techniques used by SAST tools for detecting a potential vulnerability. The first and most obvious technique is pattern matching. The tools search for some patterns in the source code that might indicate a potential vulnerability. For instance, a static analysis tool can search in source code in C the use of functions that are known to be vulnerable, such as `strcpy()`, `gets()`, `sprintf()`, etc.

A second technique that SAST tools use is to verify the control flow of a program by looking at the source code. Control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. The control flow of a program can be represented as a graph, where each node is one basic block or instruction and the transitions are the different paths that the program might take dynamically. For example, one instruction can be a conditional that according to a given condition might execute the *if* or *else* part.

The last most common technique used by SAST tools is to transform the source code into a set of data structures that represent the code known as a program model. SAST tools that operate on source code begin by transforming the code into a series of tokens by discarding all unimportant features of code such as white spaces and comments. The stream of tokens is then translated into a parse tree by the parser. The parser matches the token stream using a context-free grammar, where the grammar consists of a set of productions that describe the symbols in the language [7]. It is known that checking this model can be expensive due to the state-space explosion [24].

SAST tools have some strengths and weaknesses [25]. SAST tools are scalable: they can be run on a big quantity of software and can be run repeatedly. They are useful not only for analysing the conformance to coding standards [6] but also for finding potential vulnerabilities automatically with high confidence, such as buffer overflows, SQL Injection Flaws, cross-site scripting [16] and so forth. Their output is easy to understand for developers by highlighting the precise source files, line numbers, and even subsections of lines that are affected.

As weaknesses, we can mention that many types of security vulnerabilities are difficult to find automatically, such as authentication problems, access control issues, insecure use of cryptography, etc. The current state of the art only allows such tools to automatically find a relatively small percentage of application security flaws. However, tools of this type are getting better. Also, one of the main drawbacks of SAST tools is that they usually give a

high number of false positives although there have been approaches that improve accuracy at expense of analysis time [1] and thus affecting scalability. Moreover, SAST tools usually cannot find configuration issues since these are not represented in the code. They also have difficulties analysing code that cannot be compiled due to, for instance, missing libraries. And finally, these tools are dependent on the programming language, so if a company works with many programming languages, it will need many different SAST tools to be able to analyse all its code.

There have been different approaches to improving the effectiveness of SAST tools. For instance, in [24] the authors combine static analysis with modelling checking, which has been found to be very effective in eliminating a significant number of false positives without being affected by the non-scalability of the model checking. Another interesting approach for improving the effectiveness of static analysis is the identification and documentation of false positive patterns [31]. The goal of the study is to understand the different kinds of false positives generated so tools can automatically determine if an error message is truly indeed a true positive, and reduce the number of false positives developers and testers must triage. The results of the study have led to 14 core false positive patterns, some of which have been confirmed with static code analysis tool developers.

Another approach for improving the results of static analysis is given by methods that leverage patterns in code to narrow in on potential vulnerabilities. These patterns may be formulated by the analyst based on domain knowledge, derived from external data such as vulnerability histories, or inferred from the code directly. This methodology is called Pattern-based vulnerability discovery [55]. One of the biggest advantages of this methodology is the use of machine learning techniques for the detection of patterns as well as for vulnerability extrapolation (i.e. finding vulnerabilities similar to a known vulnerability). Although this methodology lacks a deeper understanding of program semantics, it outperforms the analysts when tasked with identifying patterns in large amounts of data, both in terms of speed and precision. This technique can be employed together with the analyst's abilities so that he/she can guide program exploration, and make final security critical decisions. Machine learning thereby becomes an assistant technology useful in different phases of the analysis.

2.2.1 SAST tools on the market

There are many SAST tools on the market. To name a few: SonarQube, SpotBugs, Synopsis Coverity, Veracode Static Analysis, Fortify Static Code Analyser, Codacy, HCL AppScan Source, Xanitizer and Checkmarx CxSAST. In this section, we will present a brief summary of some of them.

SonarQube

SonarQube [38] is an automatic code review tool to detect bugs, vulnerabilities, and code smells in the source code. It integrates with the existing work flow to enable continuous code inspection across your project branches and pull requests. During analysis, data is requested from the server, the files provided to the analysis are analysed, and the resulting data is sent back to the server at the end in the form of a report, which is then analysed asynchronously on the server side. It has support for a large variety of programming languages, being Java and TypeScript two of them, and it also be integrated with integrated development environments (IDE).

SonarQube has as well several plugins that can be integrated with different tools, for instance, Gradle, Maven, Jenkins, Ant, etc. The ability to execute the SonarQube analysis via a regular Gradle task makes it available anywhere Gradle is available, without the need to manually download, setup, and maintain a SonarQube Runner installation. The Gradle build already has much of the information needed for SonarQube to analyse a project. By preconfiguring the analysis based on that information, the need for manual configuration is reduced significantly.

One last thing to note about SonarQube is the possibility to extend it with plug-ins.

Coverity

Coverity [43] is a static analysis (SAST) solution that helps development and security teams address security and quality defects early in the software development life cycle, track and manage risks across the application portfolio, and ensure compliance with security and coding standards, such as OWASP Top 10 and CWE Top 25. Coverity provides developers issues descriptions (categories, severities, CWE information, defect location, remediation guidance, and data flow traces) as well as issue triage and management features, within their IDE. Coverity has support for several IDEs, source code management tools, issue trackers, continuous integration tools, and application life cycle management solutions.

SpotBugs

SpotBugs [39] is a program which uses static analysis to look for bugs in Java code. SpotBugs is the successor of FindBugs, carrying on from the point where it left off. SpotBugs checks for more than 400 bug patterns. SpotBugs can be used standalone or by integrating it with other tools, including Ant, Maven, Gradle, and Eclipse. One of the advantages of SpotBugs is that it is extensible through extra detectors that can be added as plug-ins, being `fb-contrib` and `Find Security` [13] two of the most important.

One nice thing about SpotBugs is that there is a plug-in for using it with SonarQube that will add coding rules to it.

In figure 2.1 we show the GUI of SpotBugs for Maven, using the NetBeans IDE, after checking the OWASP Benchmark project, presented in section 3. We see a potential SQL problem (SQL injection), which in this case is a true positive. In the code (line 49) we see that the SQL query is built directly from a string parameter without using a prepared statement. So it is indicated and explained by the tool.

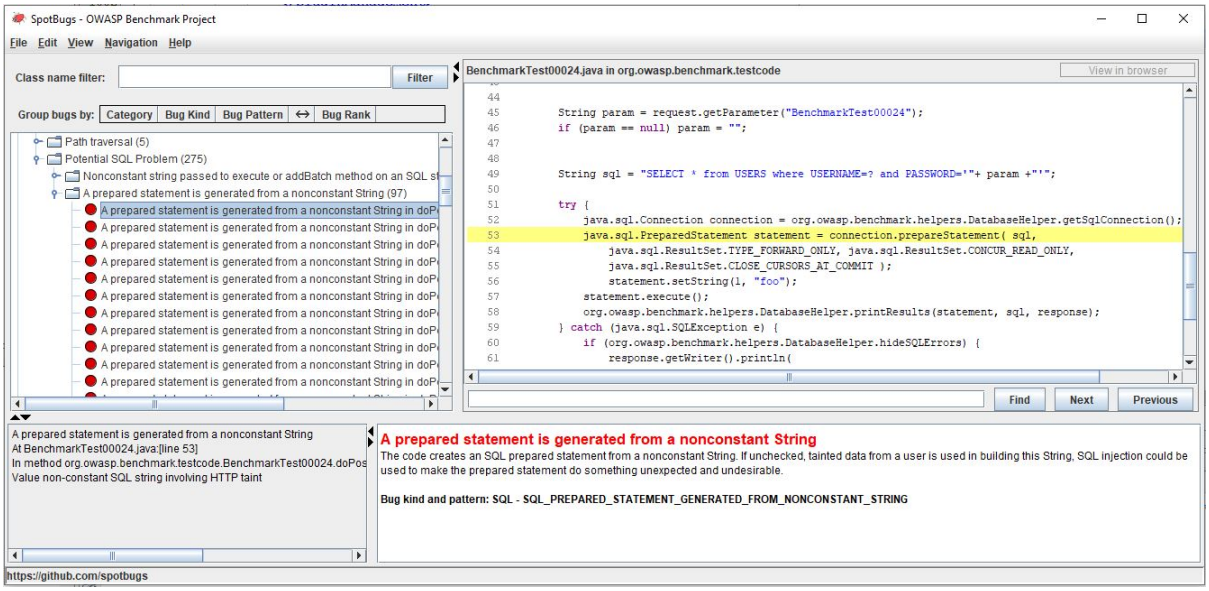

Figure 2.1: Spotbugs Maven GUI - Scan OWASP Benchmark project

AppScan Source

AppScan source [17] is a SAST tool for web application testing during the development process, with the goal of finding security issues, bugs, and anomalies before code can be committed to production environments. AppScan can be set up to run vulnerability checking tests automatically to hunt down any code vulnerabilities. Once these vulnerabilities are found, AppScan creates detailed reports with information on how to best remedy the findings.

CheckMarx

Another static code analyser is the Checkmarx CxSAST [8]. It helps in checking for errors in the source code and detecting issues with security and regulation compliance. Checkmarx SAST scans uncompiled code and does not require a complete build. For each language, the system has a list of security vulnerability issues. It can be integrated into the building of automation tools, software development, and vulnerability management.

2.3 DAST

As we know, the concept behind DAST is that it mimics a real attack. DAST tools enable the automated review of a web application by testing all the access points as they communicate through the front end. The tools simulate malicious user behaviour and emulate random actions which can be completed by complex test cases defined by an operator or interactions with third-party systems [50]. The calls, including web cryptography API, keychain, network, filesystem, SQL, as well as content providers, broadcast receivers, URI handlers, etc., will be intercepted, collected, probed, and checked for vulnerabilities to determine if every piece is behaving as it should be or if it is not part of the expected result set.

DAST testing happens once the application has advanced past its earlier life stages and has entered into a runtime state. Most DAST tools only test the exposed HTTP and HTML interfaces of web-enabled apps, but some are specifically designed for non-web protocols and data malformation, such as remote procedure calls (RPC) and session initiation protocols (SIP) [11]. DAST tools will continuously scan apps during and after development. The DAST scanners crawl, i.e. they build a map of the possible pages and access points, through a web app before scanning it. This first step allows the DAST tool to find every exposed input on pages within the app and then test each one.

The main advantage of DAST tools is that they are, for the most part, technology independent. In other words, a company can implement web applications in many different programming languages and, in principle, the same DAST tool will be able to test them while they are running. The second big advantage of DAST tools is that they can detect misconfigurations, something which is difficult or even impossible with SAST tools since most of the time the configuration is not part of the source code. DAST tools allow to test web applications in its entirety with all its interconnected parts such as web servers, proxies, databases, caches, and so on.

In addition to identifying security holes, the vulnerability scans also predict how effective countermeasures are in case of a threat or attack. A vulnerability scanning service uses pieces of software running from the standpoint of the person or organisation inspecting the attack surface in question. The vulnerability scanner uses a database to compare details about the target attack surface. The database references known flaws, coding bugs, packet construction anomalies, default configurations, and potential paths to sensitive data that can be exploited by attackers. After the software checks for possible vulnerabilities in any devices within the scope of the engagement, the scan generates a report. The findings in the report can then be analysed and interpreted to identify opportunities for an organisation to improve its security posture. Depending on the type of scan the vulnerability platform uses, various techniques and tactics will be leveraged to elicit a response

from devices within the target scope. Based on the devices' reactions, the scanner will attempt to match the results to a database and assign risk ratings (severity levels) based on those reactions [52].

DAST tools also have weaknesses. One of them is that a DAST tool, to be used to test a given web application, usually needs to be configured [36], and depending on the complexity of the web application, the configuration process can be time consuming. If we consider that a company can have many web applications, the effort can increase considerably. Another weakness of DAST tools is that even if they detect a real vulnerability, they are unable to pinpoint exactly where the problem is located as well as having difficulty following coding guidelines. Also, they may not be able to mimic an attack as someone who has internal knowledge about the application, as it may be the case with an advanced attacker [34].

2.3.1 DAST tools on the market

To list some of the DAST tools available in the market we can name: Arachni, Detectify, Acunetix, Netsparker, Managed DAST Synopsys, HCL AppScan Standard, OWASP ZAP, Subgraph Vega, etc. In this section we will present a brief summary of some of them.

AppScan Standard

AppScan Standard [17] is a dynamic application security testing tool designed for security experts and pentesters. It uses a scanning engine that automatically crawls the target app and tests for vulnerabilities. Test results are prioritised and presented in a manner that allows the operator to triage issues and home in on the most critical vulnerabilities found. Remediation is made easy using clear and actionable fix recommendations for each issue detected. In this way, continuously testing and assessing risk for web services and applications is allowed.

The users of AppScan can tailor testing to suit the needs of more elaborate test cases, by recording multi-step sequences, dynamically generating unique data and tracking a diverse set of headers and tokens.

Managed DAST Synopsys

Managed DAST Synopsys is a DAST tools part of the tools suite of Synopsis [22]. It uses automated tools to identify common vulnerabilities, such as SQL injection, cross-site scripting, security misconfigurations, and other common issues detailed in lists such as OWASP Top 10, CWE/SANS Top 25, and more. It also includes manual test cases to find vulnerabilities that cannot be

found by out-of-the-box tools, such as some vulnerabilities pertaining to authentication and session management, access control, information leakage, and more. It has as well a manual review to identify false positives and a read-out call to explain findings.

OWASP Zed Attack Proxy (ZAP)

Zed Attack Proxy (ZAP) [29] is a free open-source penetration testing tool being maintained under the umbrella of the Open Web Application Security Project (OWASP). ZAP is designed specifically for testing web applications and is both flexible and extensible. Additional functionality is freely available from a variety of add-ons in the ZAP Marketplace [56].

At its core, ZAP is what is known as a *man-in-the-middle proxy*. It stands between the tester's browser and the web application so that it can intercept and inspect messages sent between the browser and web application, modify the contents if needed, and then forward those packets on to the destination. It can be used as a stand-alone application, and as a daemon process.

As shown in figure 2.2, the easiest way of using the tool is by running a quick scan. As an example, we ran a *passive* quick scan on the site `www.uoc.edu` to test the tool.

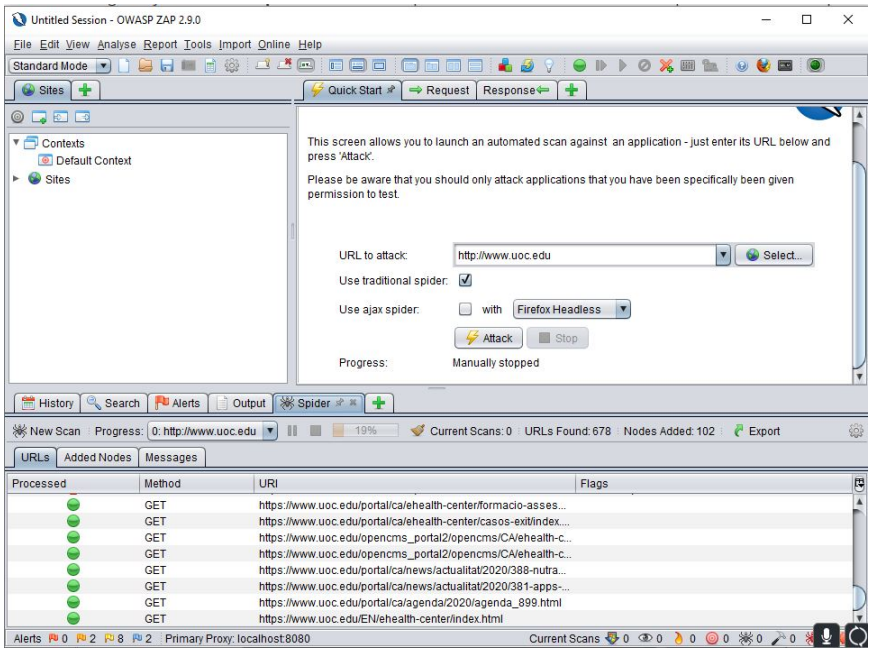

Figure 2.2: OWASP ZAP Desktop Interface - Quick scan

In figure 2.3 we show one of the warnings given by the tool while analysing `www.uoc.edu`. It is catalogued as a cross-domain misconfiguration with a medium level of risk. The reporting of the alert contains the evidence (the

offending line of code), the CWE reference number, the WASC (web application security consortium) reference number, a description, extra information, a possible solution as well as a link to an external reference.

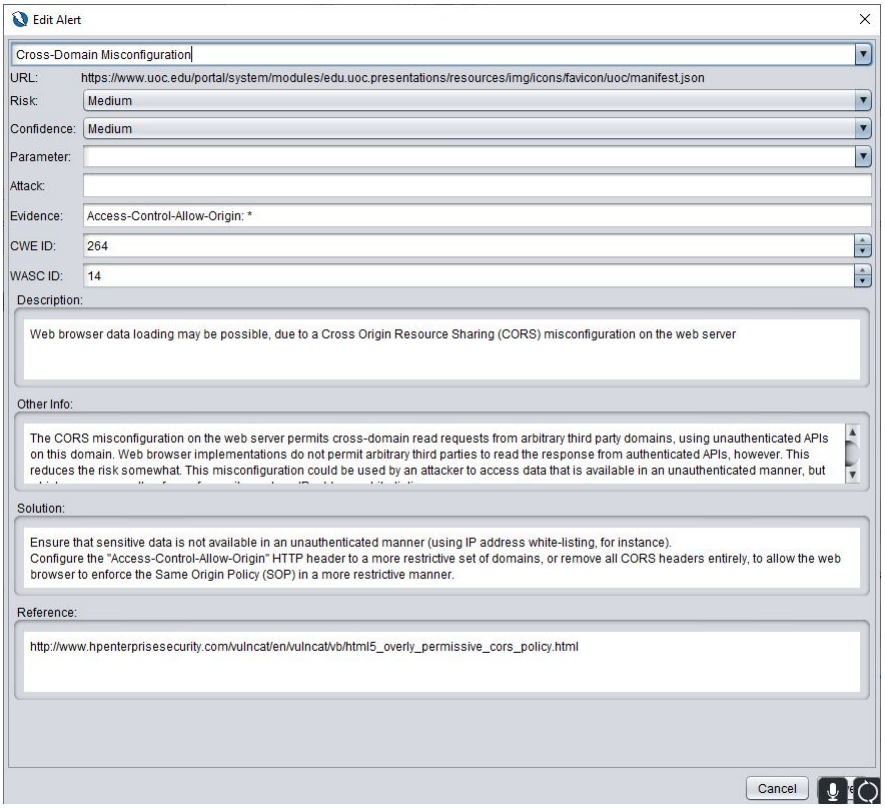

Figure 2.3: OWASP ZAP Quick scan Alert

Arachni

Arachni [5] is an Open Source, feature-full, modular, high-performance Ruby framework aimed towards helping penetration testers and administrators evaluate the security of web applications.

It is smart, it trains itself by learning from the HTTP responses it receives during the audit process and is able to perform meta-analysis using a number of factors in order to correctly assess the trustworthiness of results and intelligently identify false positives.

It is versatile enough to cover many use cases, ranging from a simple command line scanner utility, to a global high-performance grid of scanners, to a Ruby library allowing for scripted audits, to a multi-user multi-scan web collaboration platform.

In figure 2.4 we show Arachni *passively* analysing the site `www.uoc.edu`.

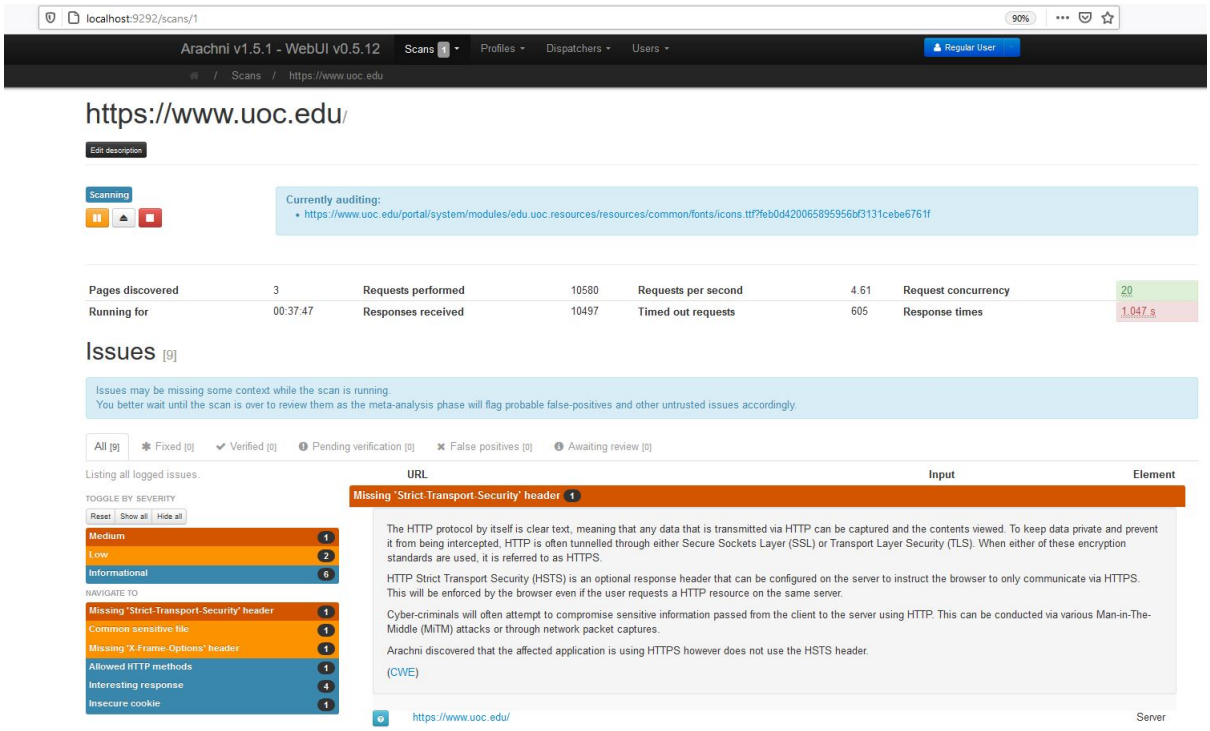

Figure 2.4: Arachni scanning www.uoc.edu

Subgraph Vega

Vega [41] is a free and open source web security scanner and web security testing platform to test the security of web applications. Vega can help you find and validate SQL Injection, Cross-Site Scripting (XSS), inadvertently disclosed sensitive information, and other vulnerabilities. It is written in Java, GUI based, and runs on Linux, OS X, and Windows.

Vega can help you find vulnerabilities such as: reflected cross-site scripting, stored cross-site scripting, blind SQL injection, remote file include, shell injection, and others. Vega also probes for TLS / SSL security settings and identifies opportunities for improving the security of your TLS servers.

In figure 2.5, we show the use of Vega when scanning the vulnerable site Google Gruyere [15]. We can see many vulnerabilities found when scanning the site, grouped in different categories, from *Info* (no risk) to *High* risk.

In figure 2.6, we show the detail of one of the vulnerabilities, namely Cross-Site scripting (XSS), found when scanning the Google Gruyere site. On the detail screen, Vega shows a summary of the vulnerability, the (http) request that detected it, the impact the vulnerability can have, the remediation (solution) and some useful links with extra information.

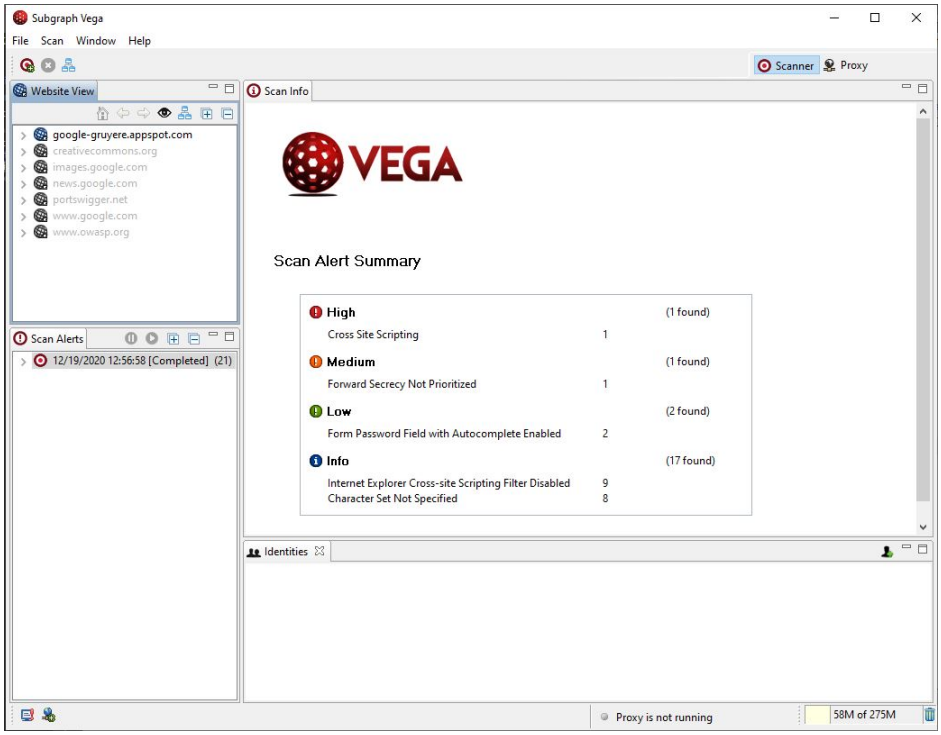

Figure 2.5: Vega scanning overview

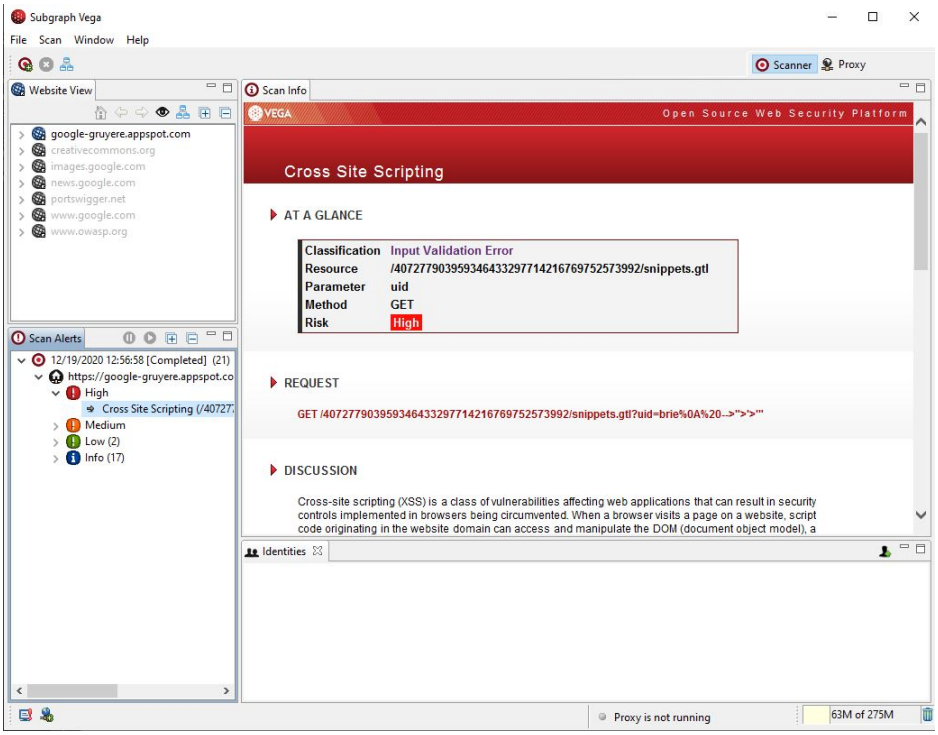

Figure 2.6: Vega scanning - detail on high risk XSS vulnerability

2.4 IAST

IAST, interactive application security testing tools, was developed as an attempt to overcome some of the limitations of SAST and DAST. Like DAST, testing occurs in real time while the application is running in a QA or test environment. Unlike DAST, however, IAST can identify the problematic line of code and notify the developer for immediate remediation [10].

As with SAST, IAST also looks at the code itself, but it does so post-build, in a dynamic environment through instrumentation of the code. IAST typically is implemented by deploying agents and sensors in the application post build. The agent observes the application's operation and analyses traffic flow to identify security vulnerabilities. It does this by mapping external signatures or patterns to source code, which allows it to identify more complex vulnerabilities.

IAST can be integrated into the CI/CD pipeline and can be automated or performed by a human tester. The biggest difference of IAST tools, with respect to SAST and DAST, is that it works from inside the application. However, IAST does not scan the entire codebase. Instead, it tests functionality only at certain points as defined by the tester, which makes it significantly faster to execute than SAST but does not provide the complete coverage SAST does.

Some of the advantages of IAST tools that we find in the literature are the low number of false positives, instant feedback from the tool as well as the fact that they are highly scalable. As weaknesses we can mention that they have limited language coverage, they usually require a mature test environment, and that they are not widely adopted [10]. Moreover, they tend to be more intrusive since they become part of the tested application or the application server.

2.4.1 IAST tools on the market

Some examples of IAST tools we can mention are Checkmarx CxIAST, Contrast Assess, Hdiv Detection and Synopsys Seeker. We review some of them.

Contrast Assess

Contrast Assess, by Contrast Security, offers interactive application security testing with elements from static application security testing and dynamic application security testing to automatically identify software vulnerabilities in real time while developers write code. Contrast Assess agents monitor code

and report from inside the application, and they claim that this enables developers to find and fix vulnerabilities without involving security experts and without specialised security expertise.

Synopsys Seeker

Synopsys claims that unlike other IAST solutions, which only identify security vulnerabilities, Seeker can also determine whether a security vulnerability (e.g., XSS or SQL injection) can be exploited. The tool then provides developers with a risk-prioritized list of verified vulnerabilities to fix in their code immediately. Seeker quickly processes hundreds of thousands of HTTP(S) requests, identifies vulnerabilities, and it is claimed that it reduces false positives to near zero. The previous would enable security teams to focus on actual verified security vulnerabilities first, greatly improving productivity and reducing business risk.

Seeker applies code instrumentation techniques (agents) inside running applications and can scale to address large enterprise security requirements. They claim that Seeker provides accurate results out of the box and does not require extensive, lengthy configuration. Seeker provides detailed vulnerability descriptions, actionable remediation advice, and stack trace information, and it identifies vulnerable lines of code.

2.5 SCA

Third-party code may save time and money, but it can also harbour some dangers if not addressed. These include security vulnerabilities, common software weaknesses (e.g. OWASP Top 10), risks related to license violations, etc. In order to know whether these third-party libraries are safe to use, we need to use a tool to analyse composition.

Software composition analysis (SCA) tools provide valuable data to security pros, legal pros, and application developers by identifying software vulnerabilities and exposing licenses for open-source components.

Advanced SCA tools automate the entire process of managing open source components, including selection, alerting on any security or compliance issues, or even blocking them from the code. They also provide comprehensive information about the open source vulnerabilities discovered so that developers can easily fix them. SCA tools can be used throughout the software development cycle, from creation to post-production [35].

Several risk factors are (or should be) analysed by SCA tools. Some of them are:

- **Component age/Outdated components**: Components may have varying degrees of age acceptance criteria. Newer versions of components

may improve quality or performance. Using components that are end-of-life or end-of-support also has some risks.

- **Known vulnerabilities**: Component analysis will commonly identify known vulnerabilities from multiple sources of vulnerability intelligence.
- **Component type**: each component can have a different degree of difficulty when upgrading or replacing. For instance, upgrading a logging framework might be very straightforward, but replacing a component for generating PDF documents might be trickier.
- **Component function**: by analysing the function of components, the SCA tool could find components with duplicate or similar functionality.
- **Component quantity**: The number of third-party and open-source components in a project should be evaluated. The operational and maintenance cost of using open source will increase with the adoption of every new component.
- **Repository trust**: Components in many software ecosystems are published and distributed to central repositories. Public repositories that have code-signing and verification requirements have some level of trust, whereas public repositories without basic countermeasures do not.
- **Provenance**: A component's provenance refers to the traceability of all changes, releases, modifications, packaging, and distribution across the entire supply chain.
- **License**: Third-party and open-source software typically have one or more licenses assigned. The chosen license may or may not allow certain types of usage, contain distribution requirements or limitations, or require specific actions if the component is modified.
- **Inherited risk**: Third-party and open-source components often have dependencies on other components. Like any component, dependencies have their own risk which is inherited by every component and application that relies on them. Components may additionally have specific runtime or environmental dependencies with implementation details not known or prescribed by the component.

2.5.1 SCA tools on the market

Some of the tools available in the market are GitLab's Dependency Scanning, WhiteSource, Synopsys Black Duck, Snyk, Sonatype Nexus Lifecycle foundation, Veracode Software Composition Analysis, OWASP dependency check, OWASP dependency track, etc. We review some of them.

OWASP dependency check

Dependency-Check [27] is a Software Composition Analysis (SCA) tool that attempts to detect publicly disclosed vulnerabilities contained within a project's dependencies. It does this by determining if there is a Common Platform Enumeration (CPE) identifier for a given dependency. If found, it will generate a report linking to the associated CVE entries.

In figure 2.7 we show the result of the analysis of the tool OWASP dependency check on the dependencies of the OWASP Benchmark project. Two critical vulnerabilities found by the tool have to do with the use of the library `log4j-1.2.17.jar`. In figure 2.8 the tool describes the vulnerabilities found for that component. The first vulnerability `CVE-2019-17571` has the description: *Included in Log4j 1.2 is a SocketServer class that is vulnerable to deserialization of untrusted data which can be exploited to remotely execute arbitrary code when combined with a deserialization gadget when listening to untrusted network traffic for log data*. The second vulnerability `CVE-2020-9488` has the description: *Improper validation of certificate with host mismatch in Apache Log4j SMTP appender. This could allow an SMTPS connection to be intercepted by a man-in-the-middle attack which could leak any log messages sent through that appender*.

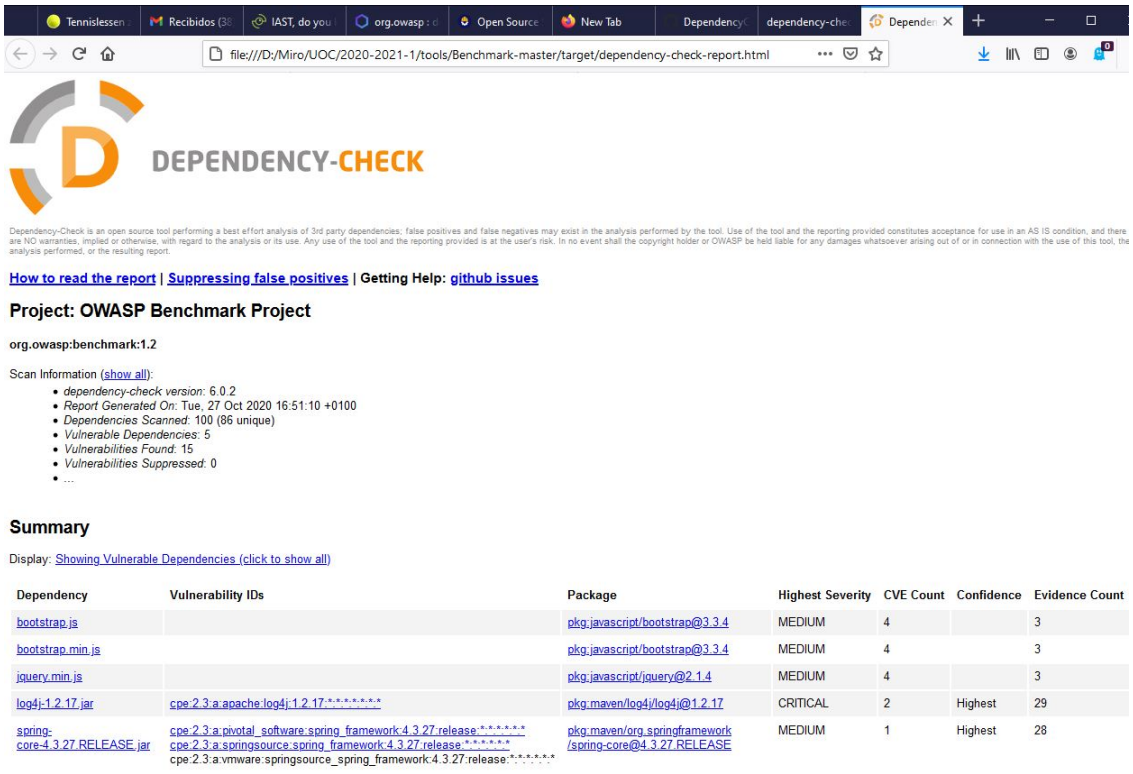

Figure 2.7: OWASP dependency check on OWASP Benchmark project

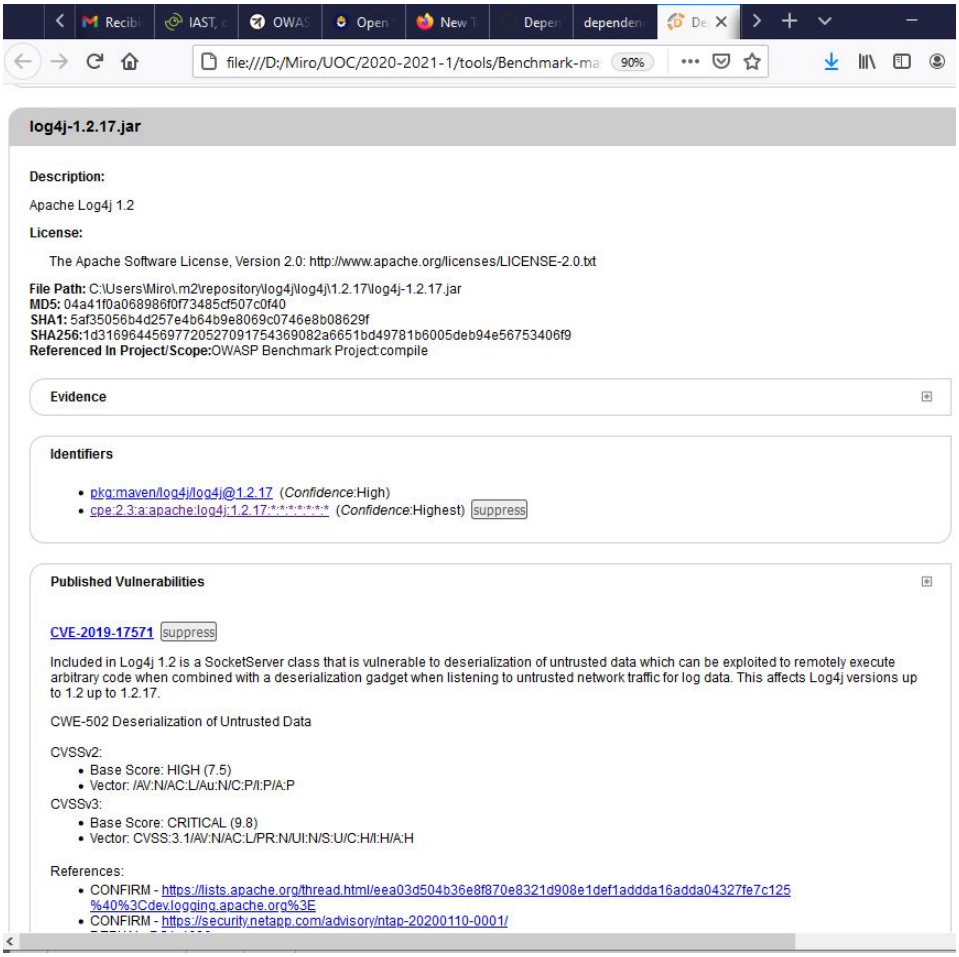

Figure 2.8: OWASP dependency check details

OWASP Dependency Track

OWASP Dependency-Track [28] is an intelligent Software Supply Chain Component Analysis platform that allows organizations to identify and reduce risk from the use of third-party and open-source components. Traditional SCA tools scan files on file system and extracts evidence with varying degrees of confidence. On the other hand, Dependency Track monitors component usage across all versions of every application in its portfolio in order to proactively identify risk across an organization.

Dependency Track takes a unique and highly beneficial approach by leveraging the capabilities of Software Bill-of-Materials (SBOM). This approach provides capabilities that, in principle, traditional SCA solutions cannot achieve. The OWASP Dependency Track platform has an API-first design and is ideal for use in Continuous Integration (CI) and Continuous Delivery (CD) environments. It tracks application, library, framework, operating system, and hardware components and identifies multiple forms of risk including components with known vulnerabilities, out-of-date components, modified components and license risk. It also integrates with multiple sources of vulnerability intelligence such as the National Vulnerability Database (NVD).

Synopsys Black Duck

Black Duck software composition analysis helps teams manage the security, quality, and license compliance risks that come from the use of open-source and third-party code in applications and containers. Black Duck Security Advisories help you avoid being caught off-guard by open-source vulnerabilities, both in development and production. And they provide the critical data necessary to prioritise vulnerabilities for remediation, such as exploit info, remediation guidance, severity scoring, and call path analysis. Black Duck also mitigates the cost and risk to intellectual property with greater insight into license obligations and attribution requirements. It integrates with build tools like Maven and Gradle to track both declared and transitive open-source dependencies in applications built in languages like Java and C#.

Snyk

One of the interesting features that Snyk provides is that it has integrated development environment (IDE) integration, which allows developers to detect vulnerable dependencies during coding to avoid future fixing efforts and save development time. Snyk gauges risk by identifying whether a vulnerable function in the used open-source library is reachable by the application or not. In this way, it prioritizes fixes based on whether vulnerable code is actually called during runtime. In the same way as Synopsys Black Duck, Snyk also possesses support for creating, customising and managing license compliance policies across your organisation.

2.6 Table: Summary of tools

	Type	Licensing
SonarQube Community Edition	SAST	Free/open source
SpotBugs	SAST	Free/open source
Synopsis Coverity	SAST	Commercial
Veracode Static Analysis	SAST	Commercial
Fortify Static Code Analyser	SAST	Commercial
Codacy	SAST	Commercial
HCL AppScan Source	SAST	Commercial
Xanitizer	SAST	Commercial
Checkmarx CxSAST	SAST	Commercial
Arachni	DAST	Free/open source
Detectify	DAST	Commercial
Acunetix	DAST	Commercial
Netsparker	DAST	Commercial
Managed DAST Synopsys	DAST	Commercial
HCL AppScan Standard	DAST	Commercial
OWASP ZAP	DAST	Free/open source
Subgraph Vega	DAST	Free/open source
Checkmarx CxIAST	IAST	Commercial
Contrast Assess	IAST	Commercial
Hdiv Detection	IAST	Commercial
Synopsys Seeker	IAST	Commercial
GitLab's Dependency Scanning	SCA	Commercial
WhiteSource	SCA	Commercial
Synopsys (Black Duck)	SCA	Commercial
Snyk	SCA	Commercial
Sonatype Nexus Lifecycle foundation	SCA	Commercial
Veracode Software Composition Analysis	SCA	Commercial
OWASP dependency check	SCA	Free/open source
OWASP dependency track	SCA	Free/open source

Table 2.1: Type and licensing of tools

Chapter 3

Benchmarks

As software engineers and security experts, we want to have a reliable and trustworthy picture of the security of our applications. Having a false sense of security due to poor security evaluation tools might be even worse than knowing that our applications are not secure at all. Indeed, in the latter case, we can act and take measures to improve the security of the applications. Therefore it becomes natural to try to evaluate the security tools in charge of measuring how secure our applications are so that we get a view as close as possible of reality.

As a consequence of what we explained above, there have been efforts in benchmarking application security tools as a way of determining their effectiveness. There are many public benchmarks available, for instance, the OWASP Benchmark project, the Web Application Vulnerability Scanner Evaluation Project (WAVSEP) and the Software Assurance Metrics and Tool Evaluation (SAMATE). We will analyse in detail these 3 benchmarks and we will present others as well that are available in the literature.

3.1 OWASP Benchmark

The OWASP Benchmark Project is a Java test suite designed to evaluate the accuracy, coverage, and speed of automated software vulnerability detection tools. Without the ability to measure these tools, it is difficult to understand their strengths and weaknesses, and compare them to each other [45].

OWASP Benchmark consists of a fully runnable open-source web application. In this webapp, we can find thousands of exploitable test cases, each case linked to a weakness in CWE, which can be analysed by any type of application security testing tool, including SAST and DAST tools. All these vulnerabilities are deliberately included in the webapp, exploitable and can be detected by the tools. The benchmark also includes dozens of scorecard generators for numerous open-source and commercial AST tools, and the set of supported tools is growing all the time.

There are several vulnerability areas or categories covered by the webapp, namely Command Injection, Weak Cryptography, Weak Hashing, LDAP Injection, Path Traversal, Secure Cookie Flag, SQL Injection, Trust Boundary Violation, Weak Randomness, XPATH Injection and XSS (Cross-Site Scripting). Each area contains a set of test cases (a Java servlet), where each test case is documented with the related CWE vulnerability, its vulnerability category, and the expected result, i.e. whether it is a true or false positive.

OWASP Benchmark tries to identify tools that do better than a random guess of whether a certain test case is a true vulnerability or not. A random guess would be for each test case to flip a coin and depending on the result, to mark the test as a true vulnerability or not. In this manner, in the end we will have around 50% of true positives detected as well as 50% of false positives. Ideally, a tool should detect as many true positives as well as few false positives as possible. In figure 3.1 from the OWASP Benchmark project, we can see a graphic representation of the previous. The dotted line indicates the tools that are as good as a random tool. The tools should be above the dotted line on the white part of the diagram, the farther to the dotted line and the closer to the 100% of the Y-axis the better the tool is in detecting vulnerabilities and avoiding false positives.

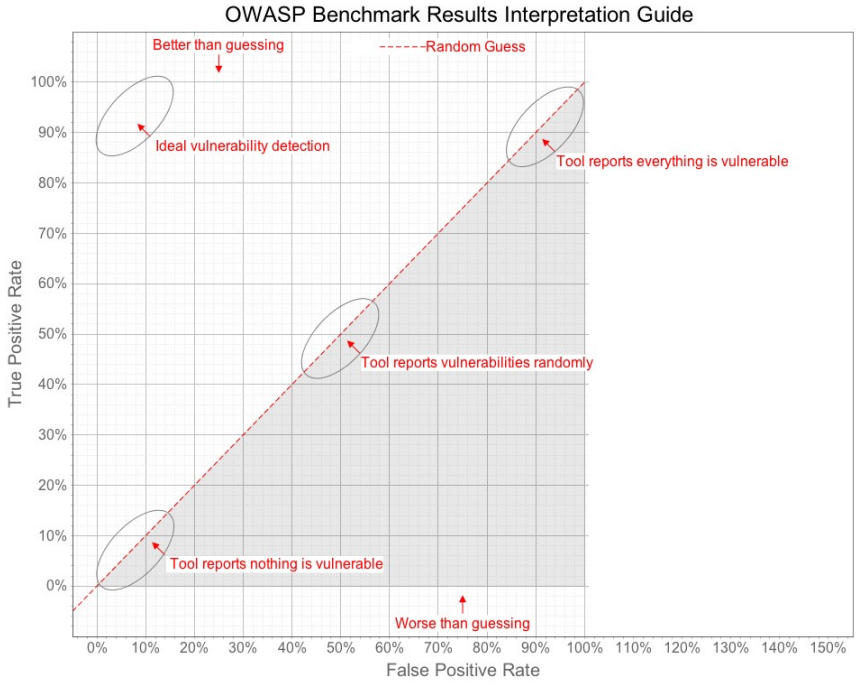

Figure 3.1: Diagram OWASP Benchmark true and false positives

There are many tools, either free/open source and commercial, that have been tested against the OWASP Benchmark by OWASP itself. Some of the

free SAST tools that have been benchmarked are PMD, SonarQube, SpotBugs, SpotBugs with the FindSecurityBugs plugin and Visual Code Grepper. Some of the commercial SAST tools that have been benchmarked are CAST Application Intelligence Platform (AIP), Checkmarx CxSAST, Julia Analyzer, Kiuwan Code Security, Micro Focus Fortify, Parasoft Jtest, Semmle LGTM, ShiftLeft SAST, Thunderscan SAST, Veracode SAST and XANITIZER. Also, there have been free DAST tools benchmarked like Arachni, OWASP ZAP and Wapiti as well as commercial ones like Acunetix Web Vulnerability Scanner, Burp Pro, HCL AppScan DAST, Micro Focus Fortify WebInspect, Netsparker, Qualys Web App Scanner and Rapid7 AppSpider. A summary of the results are shown in figure 3.2.

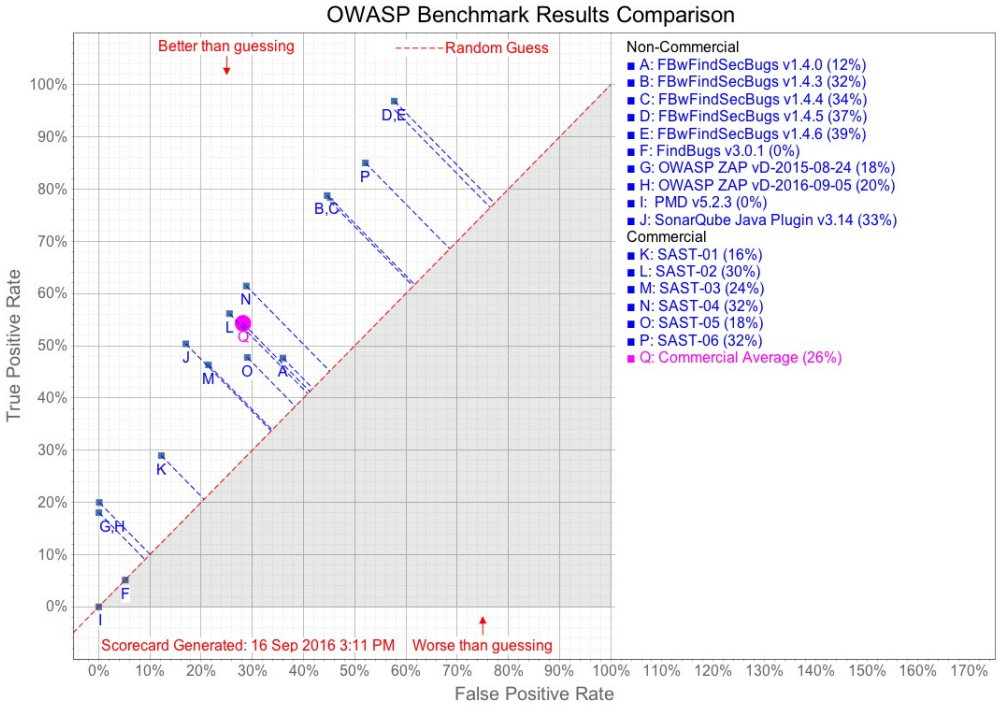

Figure 3.2: OWASP Benchmark results

The OWASP Benchmark has also been used in other works, such as in [12]. There, the Julia static analysis tool [40] is run against the OWASP Benchmark project. The Julia tool performs taint analysis by modelling explicit information flows through Boolean formulas. Taint analysis identifies every source of user data, such as form inputs and headers, and follows each piece of data through your system to make sure it gets sanitised before you do anything with it. The results of the Julia tools are shown in figure 3.3. Note that the Julia tool has been integrated into Verifysotf's CodeSonar Static Application Security Testing (SAST) platform [49].

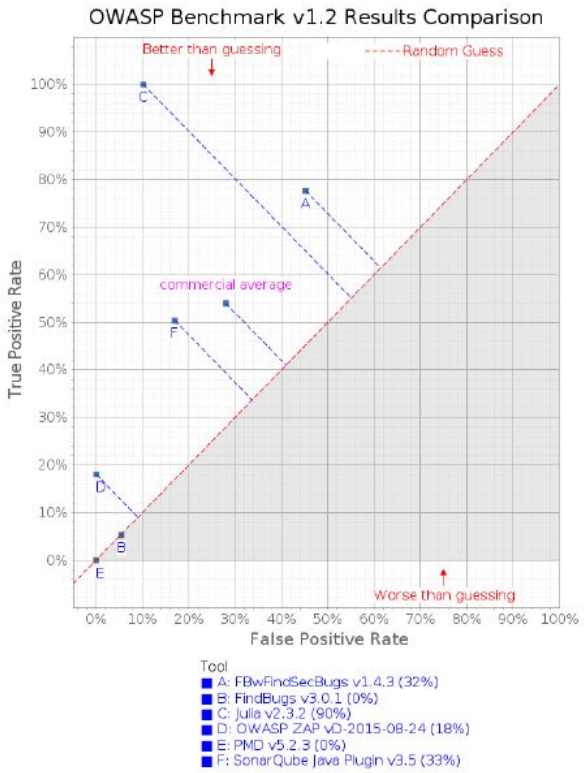

Figure 3.3: Results of the Julia tool with OWASP Benchmark

There are other tools that claim 100% detection of true positives and 0% false positives results such as RIPS [32] (SAST), Hdiv [26] (IAST) and Xanitizer [54]. However, as to our knowledge, these benchmarks have not been done by an independent external source.

3.2 WAVSEP

The Web Application Vulnerability Scanner Evaluation Project (WAVSEP) [46, 47, 48] is a vulnerable web application designed to assess the features, quality and accuracy of web application vulnerability scanners, i.e. DAST tools. It contains a collection of unique vulnerable web pages that can be used to test the various properties of web application scanners. WAVSEP contains a series of test cases with real vulnerabilities of many kinds as well as false positives.

As examples of vulnerabilities included in the WAVSEP project, we can find path traversal, remote file inclusion, cross-site scripting (XSS), SQL injection, unvalidated redirect, etc. Each of these has many test cases (or pages). For instance, there are 816 test cases for path traversal and 46 test cases for SQL injection. As for false positives, we can find XSS, SQL injections, path traversal, etc. Each of these false positive categories is located in many test cases as well.

There have been a bunch of DAST tools tested in WAVSEP. The commercial web application scanners that have been evaluated are Appspider, Netsparker, Acunetix, Burpsuite, WebInspect, WebCruiser and others. The evaluated open-source scanners are Zed Attack Proxy (ZAP), Arachni, IronWASP, WATOBO and others. In figure 3.4 we show an excerpt of the results of the last benchmark performed by the WAVSEP project.

Vulnerability Scanner	Benchmark Results
IBM AppScan	WIVET: 92% · SQLi: 100.0% · RXSS: 100.0% · LFI: 100.0% · RFI: 100.0% · Redirect: 36.67% · Backup: 5.43% · False Positive: 0.0% / 0.0% / 0.0% / 0.0% / 11.11% / 66.67% · Audit Features: 30 · Input Vectors: 17 · WebApp Scanner: ✓ · Flash Scanner: ✓ · CGI Scanner: ✓ · WebService Scanner: ✓
Acunetix WVS	WIVET: 94% · SQLi: 100.0% · RXSS: 100.0% · LFI: 94.12% · RFI: 100.0% · Redirect: 100.0% · Backup: 32.61% · False Positive: 0.0% / 0.0% / 0.0% / 0.0% / 11.11% / 0.0% · Audit Features: 29 · Input Vectors: 16 · WebApp Scanner: ✓ · Flash Scanner: ✗ · CGI Scanner: ✓ · WebService Scanner: ✓
WebInspect	WIVET: 96% · SQLi: 100.0% · RXSS: 100.0% · LFI: 91.18% · RFI: 100.0% · Redirect: 50.0% · Backup: 2.17% · False Positive: 0.0% / 0.0% / 0.0% / 0.0% / 0.0% / 0.0% · Audit Features: 29 · Input Vectors: 13 · WebApp Scanner: ✓ · Flash Scanner: ✓ · CGI Scanner: ✓ · WebService Scanner: ✓
Tinfoil Security	WIVET: 94% · SQLi: 100.0% · RXSS: 100.0% · LFI: 100.0% · RFI: 100.0% · Redirect: 100.0% · Backup: 100.0% · False Positive: 0.0% / 0.0% / 0.0% / 0.0% / 0.0% / 0.0% · Audit Features: 24 · Input Vectors: 15 · WebApp Scanner: ✓ · Flash Scanner: ✗ · CGI Scanner: ✓ · WebService Scanner: ✗
Burp Suite Professional	WIVET: 50% · SQLi: 100.0% · RXSS: 96.97% · LFI: 69.12% · RFI: 85.19% · Redirect: 76.67% · Backup: 22.28% · False Positive: 10.0% / 0.0% / 12.5% / 0.0% / 0.0% / 33.33% · Audit Features: 23 · Input Vectors: 20 · WebApp Scanner: ✓ · Flash Scanner: ✓ · CGI Scanner: ✓ · WebService Scanner: ✓

Figure 3.4: Excerpt of WAVSEP results

The interpretation of the results is left to the reader. This benchmark acts as an observation of the current features and performance of the DAST tools according to the provided test cases in WAVSEP. The authors note that the lack of support for prominent input vectors, i.e. the structure of inputs being used in the client-server communication to deliver values from the browser/mobile/client application to the web server, limits the capabilities of scanners in relevant test scenarios, particularly in various payload injection tests.

In [18] WAVSEP is used to test several tools, namely Acunetix, Burp Suite, Netsparker, AppSpider, Arachni, Vega, Wapiti, Owasp ZAP, SkipFish, IronWASP and W3af. They have selected some measures (precision, recall and F-measure) based on the number of true positives (TP), false positives (FP) and false negatives (FN). The greater the precision is, the smaller the number of false positive gets. As a result, the tool becomes more accurate in detecting the vulnerability involved. The greater the recall is, the smaller the false negative number becomes. Consequently, the tool detects the vulnerability better. They consider four vulnerability types for the tests, namely SQL

injection (SQLI), Cross-site scripting (XSS), Remote File Inclusion (FRI) and Path traversal / Local file Inclusion (LFI).

They proceed first by calculating the number of TP, FP and FN and then calculate the chosen metrics per type of vulnerability. Finally, they analysed the results and conclude that each scanner performs differently depending on the type of vulnerability. One of the most striking conclusion is that they found that there was no correlation between the cost and quality of free and commercial scanners. In figure 3.5 it is shown the results of the TP, FP and TN for each tool in each of the vulnerability types. In figure 3.6 it is shown the results of the chosen metrics for each tool in each of the vulnerability types.

Scanners	WAVSEP VULNERABILITES RESULTS											
	SQLI			XSS			RFI			LFI		
	TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP
	136		10	66		7	108		6	816		8
BurpSuite	136	0	3	62	4	0	80	28	0	496	320	1
	100%		30%	93,93%		0%	74,07%		0%	60,78%		12,5%
Wapiti	136	0	2	44	22	3	64	44	0	414	402	1
	100%		20%	66,66%		42,85%	59,25%		0%	50,73%		12,5%
Acunetix	136	0	0	66	0	0	87	21	0	292	524	0
	100%		0%	100%		0%	80,55%		0%	35,78%		0%
SkipFish	102	34	0	65	1	0	39	69	1	312	504	2
	75%		0%	98,48%		0%	36,11%		16,66%	38,23%		25%
Netsparker	136	0	3	64	2	0	57	51	0	467	349	0
	100%		30%	96,96%		0%	52,77%		0%	57,23%		0%
W3AF	81	55	3	19	47	3	13	95	1	461	355	1
	59,55%		30%	28,78%		42,85%	12,03%		16,66%	56,49%		12,5%
AppSpider	132	4	0	66	0	0	80	28	0	660	156	1
	97,05%		0%	100%		0%	74,07%		0%	80,88%		12,5%
IronWASP	136	0	5	52	14	0	106	2	0	288	528	1
	100%		50%	78,78%		0%	98,14%		0%	35,29%		12,5%
Arachni	136	0	5	63	3	0	46	62	0	162	654	0
	100%		50%	95,45%		0%	42,59%		0%	19,85%		0%
ZAP	136	0	0	63	3	0	108	0	1	590	226	0
	100%		0%	95,45%		0%	100%		16,66%	72,30%		0%
Vega	136	0	2	66	0	0	108	0	0	519	297	5
	100%		20%	100%		0%	100%		0%	63,60%		62,5%

Figure 3.5: TP, FP, TN results in Idrissi et al.

Scanners	WAVSEP Results											
	SQLI			XSS			RFI			LFI		
	Precision %	Recall %	F-measure %	Precision %	Recall %	F-measure %	Precision %	Recall %	F-measure %	Precision %	Recall %	F-measure %
BurpSuite	97,84	100	98,90	100	93,93	96,87	100	74,07	85,10	99,79	60,78	75,54
Wapiti	98,55	100	99,26	93,61	66,66	77,86	100	59,25	74,41	99,75	50,73	67,25
Acunetix	100	100	100	100	100	100	100	80,55	89,22	100	35,78	52,70
SkipFish	100	100	100	100	100	100	100	80,55	89,22	100	35,78	52,70
Netsparker	97,84	100	98,90	100	96,96	98,45	100	52,77	69,08	100	57,23	72,79
W3AF	96,42	59,55	73,62	86,36	28,78	43,17	92,85	12,03	21,30	99,78	56,49	72,13
AppSpider	100	97,05	98,50	100	100	100	100	74,07	85,10	99,84	80,88	89,36
IronWASP	96,45	100	98,19	100	78,78	88,13	100	98,14	99,06	99,65	35,29	52,12
Arachni	96,45	100	98,19	100	95,45	97,67	100	42,59	59,73	100	19,85	33,12
ZAP	100	100	100	100	95,45	97,67	99,08	100	99,53	100	72,30	83,92
Vega	98,55	100	99,26	100	100	100	100	100	100	99,04	63,60	77,45

Figure 3.6: Overview of the metrics results in Idrissi et al.

3.3 SAMATE

The SAMATE's Software Assurance Reference Dataset (SARD) Project [37] provides a set of artefacts with known software security errors and fixes for them. It also provides some *test suites* that cover many (or all) of the selected vulnerabilities. These test cases include designs, source code, binaries and other artefacts from all the phases of the software life cycle. The dataset includes *wild* (production), *synthetic* (written to test or generated), and *academic* (from students) test cases. This database will also contain real software application with known bugs and vulnerabilities. The dataset intends to encompass a wide variety of possible vulnerabilities, languages, platforms, and compilers. This will allow end users to evaluate tools and tool developers to test their methods.

In [9], the authors use a subset of the SAMATE tests, only those that are for C code and that relate to buffer overflow and *read outside the bounds of an array*, for evaluating the design of the Parfait tool. This tool is a static layered program analysis framework for bug checking, designed for scalability and precision by improving false positive rates and scale to millions of lines of code. The word *layered* means the use of a series of static program analyses that range in complexity and expense, ordered from least to more (time) expensive. The buggy statements are detected with the cheapest possible program analysis capable of detecting it, achieving in this manner better precision with the smallest runtime overhead possible. The results are quantified in terms of correctly reported, false positive and false negative rates against the NIST SAMATE *synthetic* benchmarks for C code. Figure 3.7 shows the results of the Parfait tool.

Type of Data	Raw Data	Percentage
Number of benchmarks	1182	
Number of buffer overflows	893	
Number of read outside array bounds	3	
Number of reported buffer overflows	759	85%
Number of correctly-reported buffer overflows	759	100%
Number of false positives	0	0%
Number of false negatives	58	6.5%
Number of potential bugs in potential-bug list	150	
Number of buffer overflows in potential-bug list	76	8.5%
Number of reported read outside array bounds	3	100%
Number of correctly-reported read outside array bounds	3	100%
Number of false positives	0	0%
Number of false negatives	0	0%
Average time taken per benchmark	0.2068 sec	

Figure 3.7: Parfait tool results

Besides, some SAMATE tests have also been used to perform a study to understand the coverage of the security rules in the Motorola coding standards [21]. Some of the test cases used at Motorola are developed by Motorola itself while others come from SAMATE, especially for Java and C++. After implementing the security-enhanced coding standards, supporting these new standards in a static analysis tool became the next step to focus on. The chosen tool was Klocwork because the majority of project teams at Motorola were already using it for quality purposes. However, a good percentage of these security rules were not detected by the tool. The vendor agreed to work with Motorola to improve the detection of violations to the security rules in the code.

The master thesis of Jayesh Shrestha [20] used SAMATE SARD to evaluate several static analyser security tools. There, the Juliet test cases are used for benchmarking the tools they have chosen, namely Findbugs. One of the conclusions of that work is that Findbugs was able to find the vulnerabilities that fall under certain categories like numeric errors, API abuse related, code qualitative related errors and other miscellaneous. However, it failed to detect the flaws that are related to error handling, information leaks and race conditions.

3.4 Other benchmarks

There have been as well other works that have performed benchmarking for different kinds of tools. One of them is the work of Bermejo et al. where they use the OWASP Benchmark for evaluating SAST tools [2]. The authors study the performance of seven SAST tools using a new methodology proposal and a new benchmark designed for vulnerability categories included in the known standard OWASP Top Ten project. They have chosen seven commercial and open-source SAST tools and ran them against the OWASP Top Ten Benchmark designed with the default configuration for each tool. Then they select a set of metrics that are widely accepted in the literature. Finally, they rank

the SAST tools according to their results in the benchmark and the chosen metrics.

The tools they have chosen are Xanitizer, Coverity, Checkmarx, Klocwork, Fortify, SpotBugs and FsecBugs. With respect to metrics, they have chosen the number and percentage of true positives (%TP), the number and percentage of false positives (%FP), the number of vulnerability categories for which the tool is designed, the precision (the proportion of the total TP detections and the ratio of detected vulnerabilities to the number that really exists in the code), and so on.

Some of the results are presented in the figures below. In figure 3.8 we show the results they obtained for the percentage of TP and FP. This percentage is an average of the percentages of TP and FP obtained in each vulnerability category of the OWASP Top 10, such as injection, broken authentication, etc. These results represent in an absolute manner the amount of TP and FP they have identified, however, it does not indicate, for instance, the precision of their analysis.

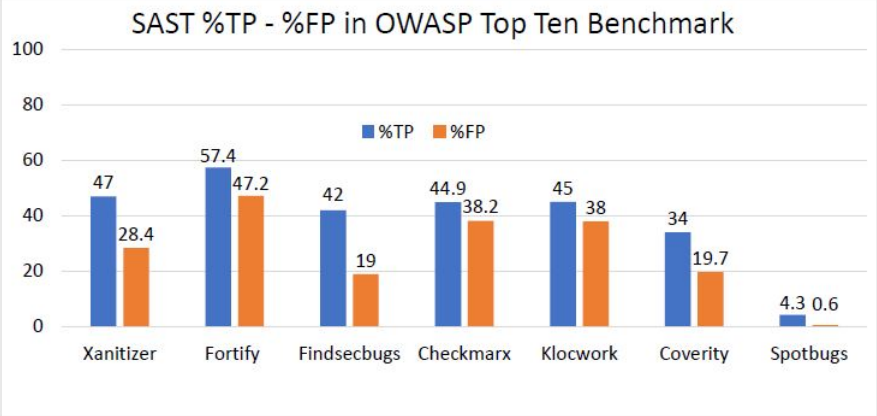

Figure 3.8: Benchmark results of Bermejo et al. for TP and FP

To have a better picture of the results, we show in figure 3.9 the numbers obtained with the different metrics used by Bermejo et al. In general, the TP ratio has a direct proportionality relationship with FP ratio. The best balance between TP and FP for a concrete tool is having a higher TP ratio with a lesser ratio of FP ratio breaking the direct proportionality relationship. The precision metric normalises TP and FP metrics penalising the ratio of TP with the ratio of FP. The average ratio of precision for all analysed tools is around the 0.6, what suggests that the SAST tools used in the experiment have a wide margin of improvement. Spotbugs and FindSecurityBugs (open source tools) have the best results in the precision metric.

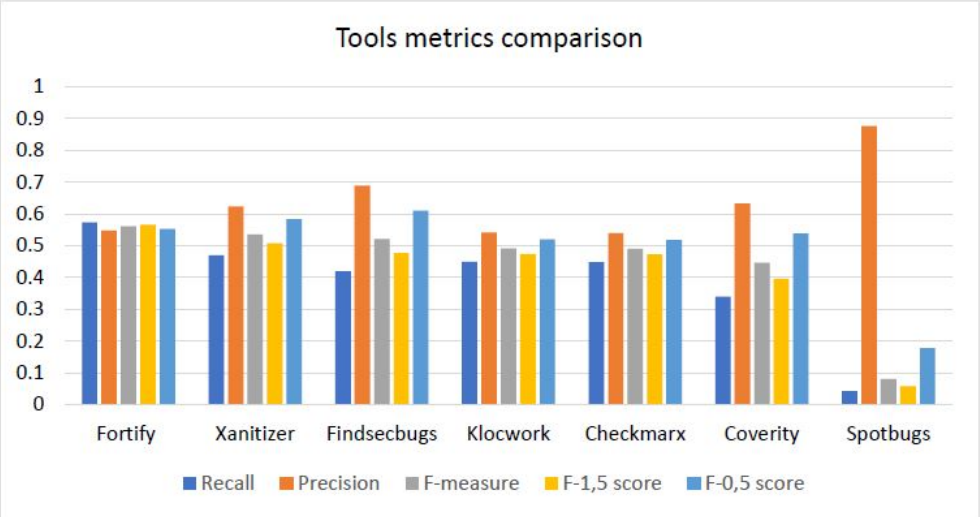

Figure 3.9: Metrics obtained for the different tools

With their approach, they obtain a ranking of the evaluated SAST tools according to adequate and wide accepted metrics applied to the results of the benchmarking. The tools are ranked having into account three distinct metrics for different degrees of importance for web applications. Five commercial SAST tools have been included in the assessment and ranked showing their results executed against the new benchmarking approach. They also suggest that using more than one SAST tool in combination can improve the TP and FP ratios.

An interesting study has been performed in [3], where the authors make a comparison between manual (normally made by a pentester) and automatic penetration testing (made by automatic DAST tools). The first step was the construction of a lab environment, which represents a typical and representative IT infrastructure of a small to medium enterprise. This environment was prepared so that it contains several vulnerabilities. The environment architecture is shown in figure 3.10. As seen on that figure, the environment consists of two hosts, one containing Kali Linux (used by the human penetration tester) and the other containing the automatic vulnerability scanners. The installed applications are popular and mostly open-source software. They include the automation server Jenkins, the content management system WordPress, the deliberately vulnerable web-application OWASP Mutillidae 2, the web development tool XAMPP and File Transfer Protocol (FTP) services installed on the Windows virtual machine (VM) and last but not least a typical Linux-based desktop computer with lots of outdated components. While Mutillidae is meant to be a vulnerable web application, the other VMs are prepared with either outdated applications, weak credentials or bad misconfiguration.

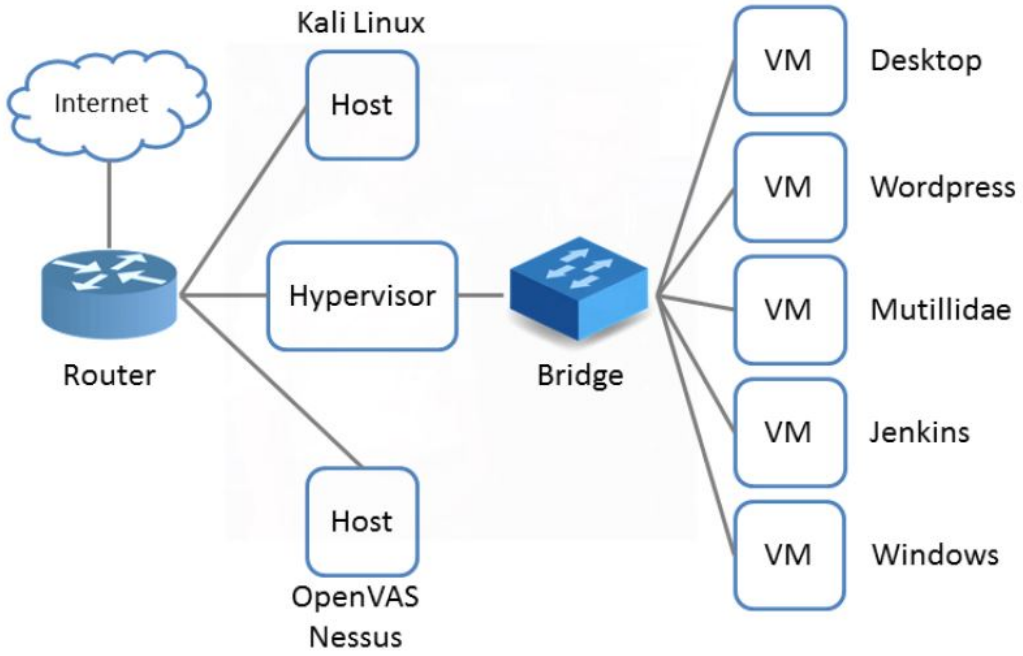

Figure 3.10: Lab environment used for penetration testing

The second step is to perform a typical penetration test by a human penetration tester without any prior knowledge of the lab environment that has been prepared. Afterwards, they used two popular vulnerability scanners to generate automated vulnerability reports in two modes, namely unauthenticated and authenticated (i.e. the tools have access to credentials). They used the proprietary software Nessus and the free software framework OpenVAS.

Finally, they validate both manually and automatically generated reports and determine error rates. The results show that the vulnerability scanner tools detect 1.5 to 3 times more vulnerabilities than the human pentester, who detected a total of 73 findings. However, to conclude that the automatic tools do a much better job than the pentester is a false conclusion because the false positives and negatives have not yet been taken into account. The human pentester has detected zero false positives and zero false negatives, i.e. he detected all the right vulnerabilities and did not give any *false alarm*. The automatic scanners detected about ten false positives and about twenty false negatives. On the other hand, scanners took about 10% of the 40 hours that the pentester needed for completing all his tasks (preparation, analysis, exploitation and final report).

The most important finding of this work is the quite higher false-negative rate of the scanners compared to the pentester, i.e. many security holes were not even detected. Therefore the results of automated methods significantly lack of quality and have less useful findings. Both manual and automated methods can complement each other, but an automated scan cannot replace

manual penetration testing nowadays. The authors propose that a good approach may be to combine both methods, which can get the best ratio of effort and results.

In [30] a methodology is shown, called Delta-Bench, for designing and building benchmarks for SAST tools. The authors start from the premise that designing a benchmark based on real-world software is a difficult task. They mention, as some of the shortcomings of the existing benchmarks, the lack of vulnerability realism, uncertain ground truth, and a large number of findings not related to analysed vulnerability. They note that purely synthetic benchmarks eliminate the above problems by isolating vulnerabilities into atomic tests that represent small applications so that each of them contains only the code relevant to a vulnerability to be tested or a deliberately inserted false positive, and some other closely related code which may be required for the vulnerable code to compile. However, one of the main motivations of this work is that it is still desirable to see how the tools perform with real-world instead of synthetic software.

Their approach for building a suitable benchmark is based on the use of large and well-documented open-source projects. Those projects contain, in their history of releases, many documented vulnerabilities that can be used for building a benchmark. Once they identify such a project, they take one of the releases with a given vulnerability as well as the release with the fix. The tools are then run against both versions and their results are compared to each other. In such a way, we can detect whether the tools have correctly detected the vulnerability or not. They tested this methodology with several SAST tools, some of them free, namely FindBugs, Jlint, OWASP LAPSE+, OWASP YASCA, PMD and SonarQube and one of them commercial, namely Fortify Static code analyzer.

In [4] the authors propose an innovative approach by combining several SAST tools to reduce the probability of vulnerabilities remaining undetected and, in this way, improve their detection rate. They selected five SAST tools that were used for finding two kinds of vulnerabilities, namely cross-site scripting and SQL injection. Those tools are RIPS, Pixy, phpSAFE, WAP and WeVerca. They studied the performance of all possible tuples of those tools, namely 10 pairs, 10 triplets, 5 quadruples and 1 quintet. With this approach, the authors provide empirically supported guidance in which of the combination produce the best results in terms of high true positives and low false positive rates. In addition, they also considered three different detection configurations of the tuples, namely raise an alarm for a vulnerability when one of the tools in the tuple detects it (1ooN), raise an alarm for a vulnerability when all of the tools in the tuple detect it (NooN), and raise an alarm for a vulnerability when the majority of the tools in the tuple detects it. As a concrete result, they show that one of the best combinations of tools of their experiment is achieved by using the diversity of Pixy, phpSAFE and WAP, especially in the 1ooN configuration.

Chapter 4

Software Security Assurance Tools Selection

So far, in chapter 2 we have given an overview of the different kinds of software security assurance tools. We have identified basically found kinds of tools, namely SAST, DAST, IAST and SCA. Afterwards, in chapter 3, we have reviewed some benchmarks methodologies that are publicly available, and some other benchmarks that have been proposed in the literature. Furthermore, we show that some independent benchmarking has been performed, i.e. benchmarking not realised by the same company offering a software security assurance product.

One of the main conclusions of the previous chapter is that software security assurance tools are aids to the development process. There is no replacement to careful coding, manual coding (in particular security) reviews, and manual penetration testing, as shown in section 3.4. They cannot be seen thus as a silver bullet that will solve all security issues in the development of software. It is still possible to write secure software without the use of any of those tools.

In this chapter we will take the steps to select the best suitable security assurance tools according to a set of predefined criteria and constraints, which will be presented below. These criteria and constraints, as explained in chapter 1, are based on the organisation of the Belgian public sector where we currently work.

4.1 Selection criteria

In this section we will define some general selection criteria for the software security assurance tools and elaborate the reasoning why we impose them.

The use of the tool must not interfere with the normal development process. As expected, the use of tools should be an aid of the development tasks of the organisation and in this way become a support of its business goals. The use of these tools should not become a bottleneck or stand in the way of an efficient software development process. Also, at a technical

level, we do not want any of the chosen tools to be included or packaged in the productive application under development. The application built for production should be the same as the one build for the tests environments. This is the only way of reliably testing aspects of the application such as performance. Moreover, we want to avoid having to build an application for a test environment with the security tool included, and afterwards building the same application without it. On the contrary, doing so can be seen as an interference in the development process and also as an error-prone process.

The tools should be ideally automatised minimising the intervention of developers. As a follow-up of the previous criterion, the chosen tools should ideally generate reports automatically without the manual intervention of the developers. For SAST tools, this can translate, for instance, in tools that can be integrated in the integrated development environment (IDE), or integrated in the continuous integration tool once the source code has been committed/ pushed in the corresponding branch of the version control system. For DAST and IAST tools, it can mean that once the source code is committed and the application deployed, the DAST/IAST tool could automatically trigger the tests on the application. For SCA tools, one option is that the dependency analysis is performed when building the application, ideally integrated in the IDE of the developers, and/or in the continuous integration tool. Of course there is a balance between the effort of configuring these tools and the degree of automation that can be obtained from the tools.

The tools must be easy to use. One extra criterion, besides having automatised software security assurance tools, is that they should be easy to use. This means that the configuration of the tools by developers should be very simple, that their execution is transparent and that they do not take an enormous amount of time for giving the results of the tests. This applies especially to SAST and SCA tools if they are integrated in an IDE; we cannot accept the developers to wait for a long period just waiting for the results of their analyses. Note that the amount of configuration is related to the degree of automation; a good balance has to be found.

The tools should be effective One of the important issues with software security assurance tools is related to the quality of the results. Tools that provide with too many false positives will produce the effect to be quickly ignored by developers due to the many results. On the other hand, we want the tools to identify all issues with the code, so that we do not miss any important vulnerability, i.e. the rate of false negatives should be low. This applies either to SAST and DAST tools. With respect to SCA tools, the results are normally less subject to discussion since the information of the vulnerable third-party open-source libraries is either scanned or kept in a database. From the results of previous chapters, we have not found any evidence that there are tools that considerably outperform the rest.

The output of these tools must be easy to understand. Developers must be able to open the results and easily identify the problems with the code. To review the results an external tool should not be needed, maybe with the exception of a web browser. The results should be detailed enough to explain the problem, to provide a possible solution, to provide extra information about the vulnerability and, if possible, to locate the vulnerability in the source code for quickly fixing it.

The tools should provide results as early as possible in the software development life cycle. It is well known that the faster a bug or problem is found in the development cycle, the cheaper it is to fix it. The detection and correction of security vulnerabilities should be to the left of the DevOps cycle.

The tools must support a certain set of technologies. It is also a requirement that the software security assurance tools must integrate with the current technologies used by the organisation. It is of no use to have, for instance, a SAST tool that can integrate with some IDEs but not the one used in the organisation. The precise description of the organisation's environment will follow in section 4.2.

The cost of the tools must be reduced or ideally free. One important aspect of the tool selection is the cost. Some organisations, especially institutions of the public sector, have budget restrictions that may be quite limiting. They usually have a limited budget foreseen for licenses. To present a proposal for several security tools that may cost several thousands of euros per year is quite a sensitive matter. This is even more so if the benchmarks that are presented in this TFM show no evidence that one single commercial software security assurance tool outperforms the rest, as shown in the benchmark performed by OWASP in the OWASP benchmark project (section 3.1). Also, in section 3.2, we saw that, in [18], one of the striking conclusions is that the authors found that there was no direct correlation between the cost and quality of free and commercial scanners. They concluded that each scanner performs differently depending on the type of vulnerability. It is unthinkable, at the level of license costs, to choose several commercial scanners, or tools in general, because some of them perform better in some areas, while the others perform better in other ones.

The tools have to be used internally. Most applications of the organisation are not accessible via internet, but internally on the intranet. It is, of course, out of the question to open the access of those applications to the internet so that a software-as-a-service (SaaS) scanner can perform automated penetration tests.

Documentation and communities. It is logical to require that the tools should have sufficient documentation for installing, configuring and using

them. Furthermore, it would be ideal to have an active community where you can find other people that have experience with the use and configuration of the tool.

4.2 Organisation's ecosystem

In this section we specify the organisation's development environment that will be considered as a reference for selecting one or many software security assurance tools.

- Programming technologies
  - Java 6, 7 & 8
  - Google Webtool Kit (GWT)
  - Angular 8 (Typescript)
  - Spring boot
  - Spring batch
  - Hibernate
- IDEs:
  - Eclipse Mars
  - Intellij IDEA Community Edition
  - Visual Studio Code
- Building tools:
  - Gradle
  - Maven repository
- Versioning control systems:
  - SVN
  - Git/Atlassian Bitbucket
- Operating Systems:
  - Developers: Windows 10
  - Servers: Linux
- CI/CD:
  - Atlassian Bamboo
- Database:
  - DB2

- MySQL

Special attention has to be put in the fact that most of the applications of the organisation are Spring boot applications. There is no external application server in which the applications are deployed. As a consequence, any tool that requires to be installed in application servers are not eligible.

4.3 Selection and exclusion of tools

For the selection of the tools there are several aspects that must be considered, as seen in section 4.1. Besides that, we need a way of evaluating which criteria can be considered more important than others. Depending each criterion's weight, the choice of the tools can be completely different.

In our case, we will consider the financial aspect the most important. The reason for this is the nature of the organisation (public sector), and the restricted budget for licenses they have. We cannot present a proposal for two or three tools that might cost eight to ten thousand euros a year each, as shown in [44] for Contrast Assess (IAST) or in [53] for Xanitizer (SAST), whereas there exist free open-source alternatives. With this decision we discard any commercial tools or paid versions of them. Only free tools or free editions of commercial tools are to be considered in the selection.

However, the strongest reason not to choose commercial tools is that we do not possess any figures to back up the possible argumentation of why the organisation should pay thousands of euros for their licenses. The risk analysis in which we measure and quantify the impact of the possible vulnerabilities that can be found by the software security assurance tools is out of the scope of this TFM. Also, the financial analysis such as the return on investment (ROI) of the use of the chosen tools are out of the scope of this TFM.

4.3.1 SAST selection

As aforementioned, we will only consider free tools. The list of SAST tools supporting the technologies listed in section 4.2 are the following: SpotBugs, Find security bugs plugin and SonarQube (SonarLite IDEs).

SpotBugs, shown in section 2.2.1, supports the Java language and it is easily integrable with Gradle. Moreover, SpotBugs has been around for many years (before known as FindBugs), it is an active project with a large users base to this day. It has a site with extensive documentation, and it has as well an eclipse plugin, but unfortunately Eclipse Mars is not supported.

Find security bugs [13], is a plugin for security audits of Java web applications. It is integrable with several IDEs, including Eclipse, IntelliJ, Android

Studio and NetBeans. It is also a quite active project nowadays, and it has a website with documentation with how to integrate and how to use.

SonarQube is a commercial tool. It has, however, one edition, namely Community Edition, which can be used for free, with some restrictions of course. On the one hand, SonarQube can be integrated with Bitbucket, and on the other hand, it has a plugin, called SonarLite, integrable with several IDEs, including Eclipse and IntelliJ. One of the shortcomings of the Community Edition of SonarQube is, however, that it does not have branch analysis. Hence, it is only possible to analyse the main (or master) branch of each project. Also, the support you might get with the Community Edition is a big question mark.

We do not see any good reason why we should not choose the three of them. Afterwards, if we see that one of the tools gives, for example, too many false positives, we might evaluate if it should be removed.

4.3.2 DAST selection

There are not many DAST tools in the market that are free. There are also some tools, like AppTrana [19], that have a *free plan* but with a SaaS model. As explained above, we will not consider these solutions because most of the applications of the organisation are internal (not accessible via internet).

We have reviewed in section 2.3.1 the Zed Attack Proxy (ZAP) tool. This tool is free and it has a website with plenty of documentation, as well as an extensive active community. Automation of ZAP tests can be achieved with the help of scripts or little programs that can be executed. It implies thus that there is a step of configuration for each site/application that has to be tested. Furthermore, ZAP has a list of add-ons that can be added to the tool to perform more specific or detailed tests.

Arachni is another free DAST tool available on the market. Unfortunately, this tool, since January 2020, is no longer maintained. The previous makes the adoption of this tool unfeasible.

There is another tool, called GoLismero. Unfortunately, we are unable to test this tool, and even less propose it for its use in the organisation, because after downloading the virus scanner detects a trojan in the downloaded file. Moreover, they claim to have plugins for IDE but when we try to download it, at this date (20/12/2020), the site gives a 404 not found error. This tool is therefore discarded.

Vega is a free DAST tool which has good documentation on its site, a similar user interface to ZAP. We have the impression that the tool is less active than ZAP, if we look at the activity on its GitHub page [42]. There, we see, for instance, that at the date of writing the current text (20/12/2020) the activity

on the last pull request dates back to March 2020.

For all the previous reasons, we see that the only two suitable tools are Vega and ZAP. We will choose ZAP over Vega because of its active community.

4.3.3 IAST selection

The only free IAST tool that we have found in the market is Contrast Community Edition. This tool can be integrated into an application server or in the application under development by including a dependency on the contrast jar file. The organisation strongly base its applications on Spring boot, so there is no external application server that can be configured once for all applications. The only option is to include a dependency of the contrast jar file in the application under development, which is not acceptable due to the selection criteria reasoned above. The reason is that we want to avoid having to build different applications, namely with or without one extra dependency, for testing or production environments. Doing so may hinder performance of the tested applications as well as creating the possibility of error if the wrong application is deployed in the wrong environment. We have the feeling that, due to the choice of SAST and DAST tools, having such a tool like Contrast does not create enough added value for the trouble.

4.3.4 SCA selection

We have found three SCA tools that are free and useful in our case. Snyk has a free version but only for open-source project, otherwise the license that will be useful for the organisation costs around two thousand a year.

The second interesting tool is OWASP Dependency Check, reviewed in section 2.5.1. Integrating this tool in the existing projects of the organisation is pretty straightforward; we only need to apply the corresponding gradle plugin, called `dependency-check-gradle`, which is publicly available. In that way, we get gradle tasks that will analyse the dependencies while building the project. The results of the tool are similar to those presented in section 2.5.1.

The last tool that can be used is the OWASP Dependency Track tool, reviewed in section 2.5.1. This tools that more than only dependency check, it also manages licensing, libraries, frameworks, operating systems, etc. This tool works at another level than the Dependency Check tool. While the Dependency Check tool works at the level of the development and build of each project, the Dependency Track tool works in a more stand-alone way, at the level of the continuous integration/continuous development. The Dependency Track tool is not integrated in the IDEs of the developers but can be integrated with the CI/CD tools of the organisation. Unfortunately, this tool

does not have support for integrating with Bitbucket. It will only be possible to link it to the maven repository (Java) and to the npm repository (Typescript/Angular).

We will choose both OWASP Dependency Check and Dependency Track. Even though we cannot take full advantage of the Dependency Track tool due to its lack of integration with the CD/CI tools of the organisation, we think the tool might be useful even though there is an effort in configuration.

4.4 Summary of selection

Selected tool	Tool Type
SonarQube Community Edition	SAST
SpotBugs	SAST
Find Security Bugs Plugin	SAST
Zed Attack Proxy (ZAP)	DAST
OWASP Dependency Check	SCA
OWASP Dependency Trace	SCA

4.5 Implementation of Quick Wins

We show in this section how we implemented some Quick Wins in the organisation with the tools that have been selected. In other words, we took the tools that needed the least configuration possible and install them or activate them. The organisation has already SonarQube Community Edition up and running but it does not use it extensively in the development process.

We would like to mention that we also thought of incorporating the OWASP dependency check tools, as it seems fairly easy to do. However, to include the dependency check we must have it first in the organisation's repository, which is a central common repository for all developers and has only libraries that are previously approved. The dependency check tool is unfortunately not part of the repository, so we were unable to include it as a quick win.

4.5.1 SonarQube

SonarQube Community Edition is installed in one of the servers. Note that, of the three projects seen on figure 4.1, only one of them has been analysed in December 2020. The other analysis date back to March and June 2020.

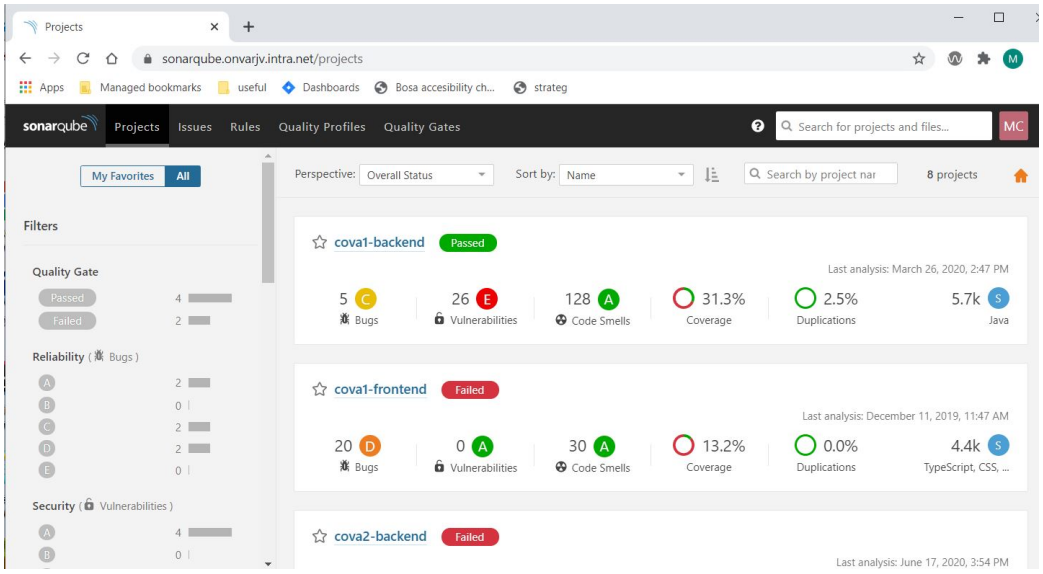

Figure 4.1: SonarQube tool analysing some projects

In figure 4.2, we show some details of the analysis of SonarQube. We see that there are several types of issues, being *Vulnerability* the most interesting for us. They also classify the issues by *Severity*, ranging from *Blocker* to *Info*. We see several issues classified as vulnerabilities and blockers, all of them in the class `ApiController` with the message *Make this method "public"*. This rule related to one of the OWASP Top 10, namely *Security Misconfiguration*. The rule says that the method should be marked as public because some aspect-oriented programming (AOP) proxies, included in the Spring framework, ignore non-public methods. Even if the method is marked with the annotation `@Secure`, this annotation will be ignored, and the method will still be called, even if the user is not authorised.

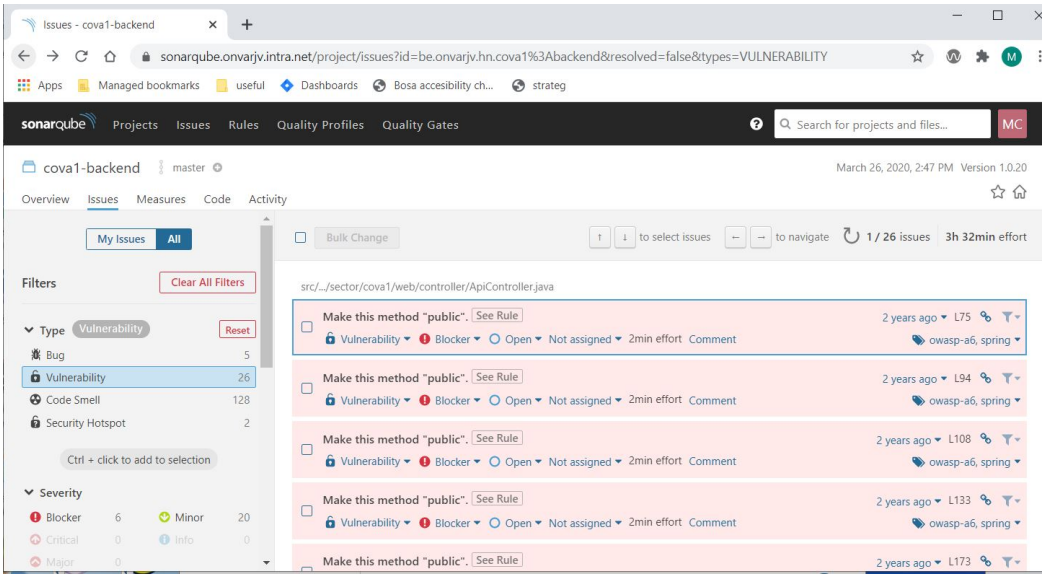

Figure 4.2: SonarQube vulnerability details of one project

However, this issue is a false positive. It is true that this might be a potential vulnerability, but in this case it is not so because the method is not annotated with any other annotation besides `@GetMapping`. In figure 4.3, we show the complete signature of one of those methods.

```
@GetMapping(path = "getCountries")
List<Country> getCountries(@RequestAttribute Cova2WebServiceCallIncoming incomingCall) {
```

Figure 4.3: Signature of method

4.5.2 SpotBugs

We have also added the SpotBugs plugin to IntelliJ for analysing source code. We have analysed the same project and we get different results. In SpotBugs, see figure 4.4, we do not get the same issue catalogued as vulnerability as the one given by SonarQube.

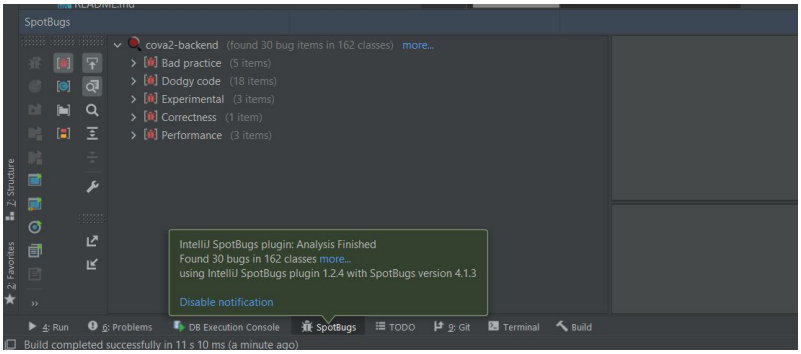

Figure 4.4: SpotBugs for Intellij

In figure 4.5, we show that there are no severe issues detected by SpotBugs. There are only grey (of concern) and yellow (troubling). Note that these categories are the two of lowest severity in SpotBugs.

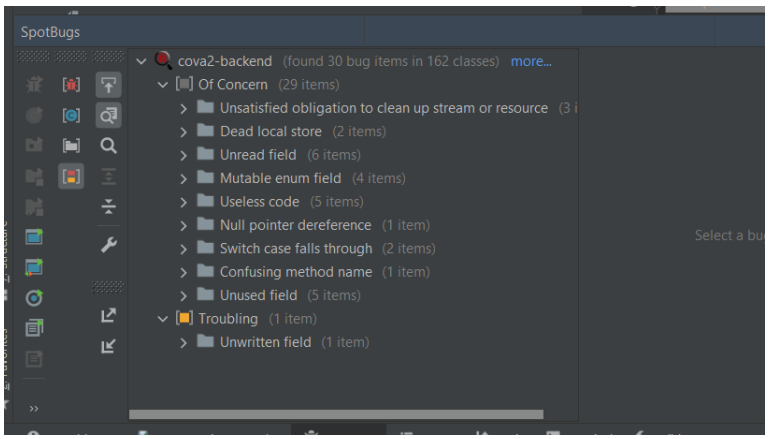

Figure 4.5: SpotBugs - Issues by severity

4.5.3 Find Security Bugs

We also installed the Find Security Bugs on IntelliJ and analysed the same project. We observe in figure 4.6 that there are only four issues, three of them related to *efficiency* and the last one related to *usability*.

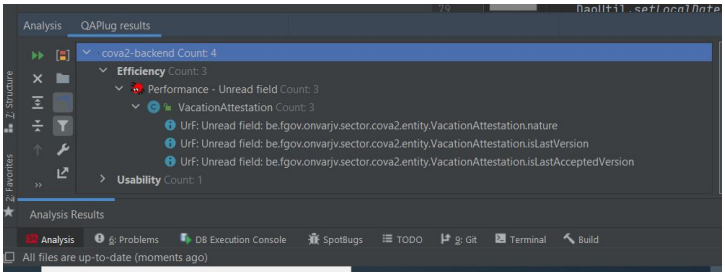

Figure 4.6: FindSecurityBugs - Issues

If we look at the issues by severity, as shown in figure 4.7, we find that there is only one critical which is related to a null pointer. In summary, no vulnerability as such has been detected by the tool.

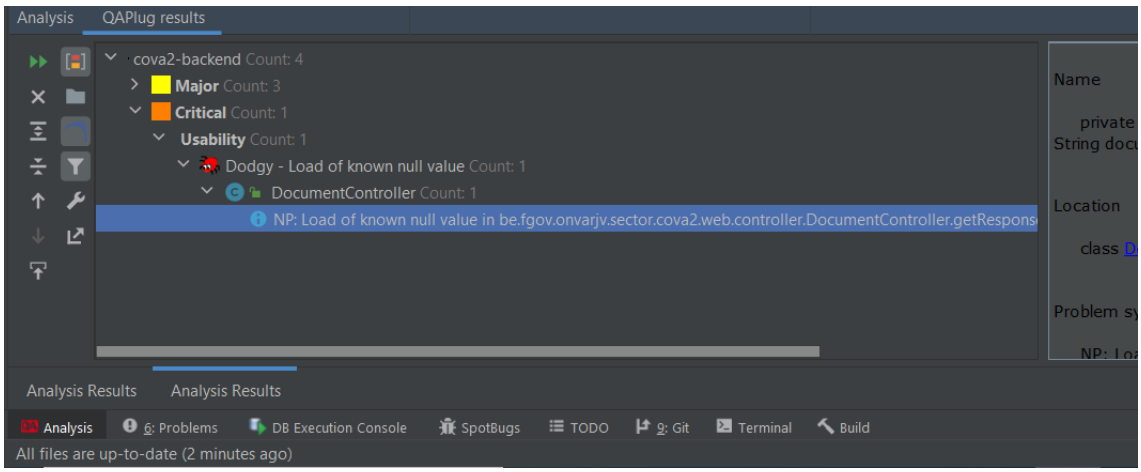

Figure 4.7: FindSecurityBugs - Issues by severity

Chapter 5

Conclusions

There are many approaches to increase the security in the development of software applications. Many of the vulnerabilities found in the software are due to errors in programming, errors in configuration, use of insecure libraries, etc. Trying to detect those problems might not be an easy task if development teams do not count with the means or helpful support given by software security assurance tools.

One objective of this TFM was to perform an extensive literature study of the different kinds of software security assurance tools that exist that can help tackle the aforementioned problem. Of course those tools are not the silver bullet; they provide help so that development teams can identify or avoid potential vulnerabilities as soon as possible in the development life cycle. However, they are not the definitive solution that will solve all security problems nor they will correct automatically all vulnerabilities. The development teams still need to take corrective actions and check the warning the tools are giving. It is therefore important that the tools are as precise and give as little false positives as possible. Another objective of this TFM is to define a set of criteria for the selection of tools, based on the current situation in the organisation where we currently work, and to make a proposal of which tools can be the most suitable to improve quality and therefore security in the development life cycle.

In chapter 1, we introduced and motivated the problem, we explained the objectives of the TFM, we showed the methodology that we used in the development of this TFM, and we presented the planning. Finally, we defined what were the most important contributions of this work.

We have reviewed, in chapter 2, the kinds of software security assurance tools that are most popular in the market. We have focused on four kinds of tools, namely SAST, DAST, IAST and SCA. In one word, SAST tools analyse source code in a non-running state; DAST tools perform passive and active penetration tests on application in a running-state; IAST tools incorporate agents in the application that can analyse bytecode as well as performing some kind of dynamic analysis; and SCA focuses on the analysis of dependencies that might be vulnerable, or in general, that might pose a risk. Moreover, for each kind of tools, we mentioned the most relevant software

products in the market, and we gave a brief summary of some of them.

In chapter 3, we presented a summary of the three most well-known benchmark methodologies/frameworks for SAST and DAST tools, namely the OWASP benchmark project, the WAVSEP project and SAMATE. We have also reviewed the many benchmarks performed with those methodologies in the literature. Afterwards, we reviewed literature of other benchmarks performed using their own custom methodology. Two of the most important conclusions of this chapter is that each tool performs better in different areas, and that there is no direct correlation between a tool being commercial and a better effectiveness.

In chapter 4, we defined the criteria that we were going to follow for selecting the most suitable software security assurance tools. Those criteria encompass aspects varying from financial (i.e. what is the cost of implementing or integrating the selected tools), technical (i.e. how the selected tools integrate with the rest of the software of the organisation and how it fits in the development life cycle) and effectiveness (i.e. how well the tools performs in their support to the development team). Based on those criteria we have chosen a set of tools, and implemented the ones that were considered to be Quick Wins.

Bibliography

[1] G. Agosta et al. “Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution”. In: *2012 Ninth International Conference on Information Technology - New Generations*. 2012, pp. 189–194.

[2] J. R. Bermejo Higuera et al. “Benchmarking approach to compare web applications static analysis tools detecting owasp top ten security vulnerabilities”. In: *Computers, Materials and Continua*. Vol. 64. 3. 2020, pp. 1555–1577.

[3] Saed Alavi, Niklas Bessler and M. Massoth. “A Comparative Evaluation of Automated Vulnerability Scans Versus Manual Penetration Tests on False-negative Errors”. In: *CYBER 2018, The Third International Conference on Cyber-Technologies and Cyber-Systems*. Athens, Greece: IARIA, 2018, pp. 1–6.

[4] A. Algaith et al. “Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools”. In: *2018 14th European Dependable Computing Conference (EDCC)*. Iasi, Romania, 2018, pp. 57–64.

[5] *Arachni scanner*. Oct. 2020. url: https://www.arachni-scanner.com/ (visited on 20/10/2020).

[6] Q. Ashfaq, R. Khan and S. Farooq. “A Comparative Analysis of Static Code Analysis Tools that check Java Code Adherence to Java Coding Standards”. In: *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*. 2019, pp. 98–103.

[7] M. Lal Das C. Chahar V. Singh Chauhan. “Code analysis for software and system security using open source tools”. In: *Information security journal: A global perspective* 21 (2012), pp. 346–352. url: http://link.aip.org/link/?RSI/72/4477/1.

[8] *Checkmarx CxSAST*. Oct. 2020. url: https://www.checkmarx.com/products/static-application-security-testing (visited on 20/10/2020).

[9] C. Cifuentes and B. Scholz. “Parfait: designing a scalable bug checker”. In: *SAW ’08: Proceedings of the 2008 workshop on Static analysis*. Tucson, Arizona: ACM, 2008, pp. 4–11.

[10] *Does IAST Fit Into Your AppSec Program?* Oct. 2020. url: https://resources.whitesourcesoftware.com/blog-whitesource/iast-interactive-application-security-testing (visited on 20/10/2020).

[11] *Dynamic application security testing (DAST)*. Oct. 2020. url: https://searchapparchitecture.techtarget.com/definition/dynamic-application-security-testing-DAST (visited on 20/10/2020).

[12] P. Ferrara, E. Burato and F. Spoto. "Security Analysis of the OWASP Benchmark with Julia". In: *In Proceedings of the First Italian Conference on Cybersecurity (ITASEC17)*. 2017.

[13] *Find Security Bugs plugin for SpotBugs*. Oct. 2020. url: https://find-sec-bugs.github.io/ (visited on 20/10/2020).

[14] E. Fong and V. Okun. "Web Application Scanners: Definitions and Functions". In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. 2007, 280b–280b.

[15] *Google Gruyere vulnerable site*. Oct. 2020. url: https://google-gruyere.appspot.com/ (visited on 20/10/2020).

[16] M. K. Gupta, M. C. Govil and G. Singh. "Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey". In: *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*. 2014, pp. 1–5.

[17] *HCL AppScan*. Oct. 2020. url: https://www.hcltechsw.com/products/appscan (visited on 20/10/2020).

[18] S. Idrissi et al. "Performance evaluation of web application security scanners for prevention and protection against vulnerabilities". In: *International Journal of Applied Engineering Research* 12 (Jan. 2017), pp. 11068–11076.

[19] *Indusface Apptrana*. Oct. 2020. url: https://www.indusface.com/web-application-scanning.php (visited on 20/10/2020).

[20] Shrestha J. "Static Program Analysis". MA thesis. The address of the publisher: Uppsala University, Sept. 2013. url: http://www.diva-portal.org/smash/get/diva2:651821/FULLTEXT01.pdf (visited on 20/10/2020).

[21] R Krishnan, Margaret Nadworny and Nishil Bharill. "Static Analysis Tools for Security Checking in Code at Motorola". In: *Ada Letters* XXVIII.1 (Apr. 2008), pp. 76–82.

[22] *Managed Dynamic Application Security Testing - Synopsis*. Oct. 2020. url: https://www.synopsys.com/software-integrity/application-security-testing-services/dynamic-analysis-dast.html (visited on 20/10/2020).

[23] *MITRE, Common Weakness Enumeration*. Oct. 2020. url: https://cwe.mitre.org/data/index.html (visited on 18/10/2020).

[24] T. Muske and U. P. Khedker. "Efficient Elimination of False Positives using Static Analysis". In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015.

[25] *OWASP - The Open Web Application Security Project*. Oct. 2020. url: https://owasp.org/ (visited on 20/10/2020).

[26] *OWASP Benchmark - Hdiv Detection (IAST)*. Oct. 2020. url: https://hdivsecurity.com/owasp-benchmark (visited on 20/10/2020).

[27] *OWASP dependency check project*. Oct. 2020. url: https://owasp.org/www-project-dependency-check/ (visited on 20/10/2020).

[28] *OWASP Dependency Track*. Oct. 2020. url: https://dependencytrack.org/ (visited on 20/10/2020).

[29] *OWASP Zed Attack Proxy (ZAP)*. Oct. 2020. url: https://www.zaproxy.org/ (visited on 20/10/2020).

[30] I. Pashchenko, S. Dashevskyi and F. Massacci. "Delta-Bench: Differential Benchmark for Static Analysis Security Testing Tools". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Vol. 1. Toronto, Canada, 2017, pp. 163–168.

[31] Z. P. Reynolds et al. "Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools". In: *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*. 2017, pp. 55–61.

[32] *RIPS Scores a Perfect 100% at OWASP Benchmark*. Oct. 2020. url: https://blog.ripstech.com/2020/owasp-benchmark/ (visited on 20/10/2020).

[33] Rebecca Russell et al. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Dec. 2018, pp. 757–762.

[34] *SAST vs DAST: What is the right choice for application security testing?* Oct. 2020. url: https://codedx.com/blog/sast-vs-dast-tools/ (visited on 20/10/2020).

[35] *SAST vs SCA: It's like comparing apples and oranges*. Oct. 2020. url: https://resources.whitesourcesoftware.com/blog-whitesource/sast-vs-sca (visited on 18/10/2020).

[36] *Security Tools Benchmarking - A blog dedicated to aiding pen-testers in choosing tools that make a difference*. Oct. 2020. url: http://sectooladdict.blogspot.com/2017/05/dast-vs-sast-vs-iast-modern-ssldc-best.html (visited on 20/10/2020).

[37] *Software Assurance Metrics And Tool Evaluation (SAMATE) - Software Assurance Reference Dataset Project (SARD)*. Oct. 2020. url: https://samate.nist.gov/SRD/testsuite.php (visited on 20/10/2020).

[38] *SonarQube*. Oct. 2020. url: https://www.sonarqube.org/ (visited on 20/10/2020).

[39] *SpotBugs*. Oct. 2020. url: https://spotbugs.github.io/ (visited on 20/10/2020).

[40] Fausto Spoto. “The Julia Static Analyzer for Java”. In: *Static Analysis*. Ed. by Xavier Rival. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 39–57.

[41] *Subgraph Vega*. Oct. 2020. url: https://subgraph.com/vega/index.en.html (visited on 20/10/2020).

[42] *Subgraph Vega - Github Pull Requests*. Oct. 2020. url: https://github.com/subgraph/Vega/pulls (visited on 20/10/2020).

[43] *Synopsis Coverity*. Oct. 2020. url: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html (visited on 20/10/2020).

[44] *The Contrast Assess Cost Advantage - Application Security Testing Costs Compared*. Oct. 2020. url: https://www.contrastsecurity.com/hubfs/docs/WPCostAdvantage121916.pdf (visited on 20/10/2020).

[45] *The OWASP Benchmark Project*. Oct. 2020. url: https://owasp.org/www-project-benchmark/ (visited on 20/10/2020).

[46] *The Web Application Vulnerability Scanner Evaluation Project*. Oct. 2020. url: https://code.google.com/archive/p/wavsep/ (visited on 20/10/2020).

[47] *The Web Application Vulnerability Scanner Evaluation Project*. Oct. 2020. url: https://github.com/sectooladdict/wavsep (visited on 20/10/2020).

[48] *The Web Application Vulnerability Scanner Evaluation Project*. Oct. 2020. url: http://sectooladdict.blogspot.com/2017/11/wavsep-2017-evaluating-dast-against.html (visited on 20/10/2020).

[49] *Verifysotf's CodeSonar*. Oct. 2020. url: https://www.verifysoft.com/en_grammatech_codesonar.html (visited on 20/10/2020).

[50] *What is DAST?* Oct. 2020. url: https://www.sqreen.com/web-application-security/what-is-dast#how-do-dast-tools-work (visited on 20/10/2020).

[51] *What is IAST? Interactive Application Security Testing*. Oct. 2020. url: https://www.veracode.com/security/interactive-application-security-testing-iast (visited on 20/10/2020).

[52] *What is vulnerability scanning, and how does it work?* Oct. 2020. url: https://www.redlegg.com/blog/what-is-vulnerability-scanning-and-how-does-it-work (visited on 19/10/2020).

[53] *Xanitizer*. Oct. 2020. url: https://www.rigs-it.com/xanitizer/ (visited on 20/10/2020).

[54] *Xanitizer News*. Oct. 2020. url: https://www.xanitizer.com/news/ (visited on 20/10/2020).

[55] Fabian Yamaguchi. “Pattern-Based Vulnerability Discovery”. PhD thesis. Georg-August-Universität Göttingen, 2015.

[56] *ZAP marketplace*. Oct. 2020. url: https://www.zaproxy.org/addons/ (visited on 20/10/2020).