Integration of automated code analysis tools

Víctor Morga Marchal

Cybersecurity and privacy
Data analysis

Supervisor
Joan Caparrós Ramírez
Professor
Andreu Pere Isern Deyà

Submission Date
01/2023

Universitat Oberta
de Catalunya

To all my friends and family, without them I would not be where I am now.

DATASHEET

Title:	*Integration of automated code analysis tools*
Author:	*Víctor Morga Marchal*
Supervisor:	*Joan Caparrós Ramírez*
Professor:	Andreu Pere Isern Deyà
Submission date (mm/yyyy):	*01/2023*
Degree:	Cybersecurity and privacy
Field:	*Data analysis*
Language:	*English*
Keywords:	*Automated, code, analysis*

Abstract

This paper aims to research and explain how to improve the security of software products with the use of automated code analysis tools and their integration in the software development lifecycle in an efficient way.

The automated code analysis tools are categorized in four groups: SAST, DAST, IAST and SCA. This paper explains each of these groups, give examples of current products and show in which deployment tiers are integrated.

Once the technology is explained, a real software development environment is set up, consisting of a source code repository, a continuous integration / continuous delivery server and the automated code analysis tools.

Finally, we integrate all the components to show how can any company or developer take advantage of these tools, making their final product more secure.

Index

Figures

1. Introduction

1.1. Context

The number of cyber attacks has increased over the last few years, causing data breaches and huge financial loss to companies (1). The attacks are performed by cyber criminals using a variety of techniques, they can target people through social engineering or exploit a vulnerability in the systems directly. These vulnerabilities occur due to many reasons: bad configurations, outdated components or unsecure code among others can open a door in our systems to an attacker.

In order to produce software that is secure, companies need to implement security measures through all the software development lifecycle. One of these measures is the use of automated code analysis tools. These tools can be used through all the development phases to detect code that can be vulnerable.

This paper aims to explain the technology that is used under this category, how it works and how it can be implemented in software development lifecycle. With these tools we will be able to reduce the costs of security testing and find vulnerabilities and legal risks regarding third party components licenses before releasing the product to the public.

The integration of these tools during the product development has other benefits: In DevSecOps, the implementation of security measures through all the development lifecycle instead of only in the final states is known as pushing left the application security. Pushing left allows us to find vulnerabilities in early stages of the development, which allow the developers to solve the problem instantly instead of doing a huge code refactoring during the late stages, thus saving time and money to the company.

1.2. Goals

As stated in the previous section, the goal of this project is to detect vulnerabilities and license issues as soon as possible in software development using automated code analysis tools, which will save time and money for the companies. In order to achieve this goal, we have three main objectives:

- Enumerate and explain how the technology that is categorized under automated code analysis tools works and what benefits provide to the companies. These technologies are: SAST, DAST, IAST and SCA.

- List and compare the pros and cons of some automated code analysis tools that are currently available in the market and can be implemented in a software company.

- How to integrate the automated code analysis tools in the software development lifecycle to enhance the security in each step.

1.3. Methodology

For this project we are going to follow an agile methodology. The project is going to be divided into three deliveries or sprints.

In the first sprint we are going to research the different technologies that are categorized under automated code analysis tools (SAST, DAST, IAST and SCA). In this sprint we are going to explain what they are, what their purpose is, how they work, how they can be implemented in our development lifecycle and give some examples of current solutions that are available in the market.

In the second sprint we are going to install a software development environment with the current CI/CD (continuous integration/continuous delivery) technologies, such as source code repositories, automated building tools and automated deployment tools. In this environment we are going to integrate the automated code analysis tools explained in the previous delivery.

The third sprint will be used to optimize and finish the setup of the automated code analysis tools. After finishing the implementation, we will write the conclusions of this project and start elaborating the demo and the presentation for the defense

1.4. Tasks

This paper is going to be divided into three phases: Research, implementation and optimizations & conclusions.

1.4.1. Research phase

This phase will be used to study the different technologies that we are going to study, what they do, what their purpose is, how they work and how they are implemented.

1.4.1.1. Static application security testing (SAST) research

SAST is used for the analysis of the code before it is compiled, also known as white box testing. The tasks in this chapters are:

- Investigate what is SAST and how it works
- Investigate current solutions that implement SAST and list their pros and cons
- Investigate how we can integrate SAST in the development lifecycle

1.4.1.2. Dynamic Application Security Testing (DAST) research

In this section we will study how to test an application after it is compiled through simulated attacks.

- Investigate what is DAST and how it works.
- Investigate current solutions that implement DAST and list their pros and cons.
- Investigate how we can integrate DAST in the development lifecycle.

1.4.1.3. Interactive application security testing (IAST) research

IAST tools test an application using software instrumentation:

- Investigate what is IAST and how it works.
- Investigate current solutions that implement IAST and list their pros and cons.
- Investigate how we can integrate IAST in the development lifecycle.

1.4.1.4. Software Composition Analysis (SCA)

SCA is the technology that analyzes the dependencies in our project. The tasks are:

- Investigate what is SCA and how it works.
- Investigate current solutions that implement SCA and list their pros and cons.
- Investigate how we can integrate SCA in the development lifecycle.

1.4.2. Implementation phase

In this phase we will install and setup a CI/CD environment in which we will implement the studied tools of the previous phase:

- Install and set up the CI/CD environment (source code repository, automated building tools, automated deployment tools,...).
- Install and set up the chosen SAST tool.
- Install and set up the chosen DAST tool.
- Install and set up the chosen IAST tool.
- Install and set up the chosen SCA tool.

1.4.3. Optimization and conclusion phase

In this phase we will optimize and finish the setup of the automated code analysis tools. After the whole integration is completed, we will write the conclusion of this project

- Optimize and finish the setup of the SAST tool.
- Optimize and finish the setup of the DAST tool.
- Optimize and finish the setup of the IAST tool.
- Optimize and finish the setup of the SCA tool.
- Write conclusions.

1.5. Planning

For this project we have dedicated the number of hours required for 12 ECTS. The planning for this project is explained by the following table and Gantt diagram. The weekends are included due to the student employment status, but the national holidays are excluded.

Task	Start date	End date	Hours
1 Introduction	3/10/2022	11/10/2022	18
1.1. Planning	3/10/2022	11/10/2022	5.5
1.2. Introduction	3/10/2022	11/10/2022	6
1.3. State of art	3/10/2022	11/10/2022	6
2. Research phase	12/10/2022	8/11/2022	70
2.1. SAST research	12/10/2022	19/10/2022	17.5
2.2. DAST research	19/10/2022	26/10/2022	17.5
2.3. IAST research	26/10/2022	1/11/2022	17.5
2.4. SCA research	1/11/2022	8/11/2022	17.5
3. Implementation phase	9/11/2022	6/12/2022	92
3.1. Implementation of CI/CD environment	9/11/2022	23/11/2022	28
2.1. SAST tool installation & setup	16/11/2022	30/11/2022	16
2.2. DAST tool installation & setup	23/11/2022	6/12/2022	16
2.3. IAST tool installation & setup	23/11/2022	6/12/2022	16
2.4. SCA tool installation & setup	30/11/2022	6/12/2022	16
4. Optimization and conclusion phase	7/12/2022	10/01/2023	89
4.1. Optimization of SAST tool	7/12/2022	3/01/2023	20
4.2. Optimization of DAST tool	7/12/2022	3/01/2023	20
4.3. Optimization of IAST tool	7/12/2022	3/01/2023	20
4.4. Optimization of SAST tool	7/12/2022	3/01/2023	20
4.5. Write conclusions	3/01/2023	10/01/2023	9
5. Record video	11/01/2023	17/01/2023	15
6. Create presentation	18/01/2023	22/01/2023	15
7. Defend dissertation	23/01/2023	27/01/2023	1
		Total	300

Gantt diagram:

Task	Week																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1 Planning and introduction																	
1.1. Planning																	
1.2. Introduction																	
1.3. State of art																	
2. Research phase																	
2.1. SAST research																	
2.2. DAST research																	
2.3. IAST research																	
2.4 SCA research																	
3. Implementation phase																	
3.1.Implementation of CI/CD environment																	
3.2. SAST tool installation & setup																	
3.3. DAST tool installation & setup																	
3.4, IAST tool installation & setup																	
3.5. SCA tool installation & setup																	
4. Optimization and conclusion phase																	
4.1. Optimization of SAST tool																	
4.2. Optimization of DAST tool																	
4.3. Optimization of IAST tool																	
4.4. Optimization of SCA tool																	
4.5. Write conclusions																	
5. Record video																	
6. Create presentation																	
7. Defend dissertation																	

1.6. State of art

For this introduction, a research has been conducted to know what is the current situation of the integration of automated code analysis tools in companies.

1.6.1. CI/CD servers

Jenkins software is the most popular among tech companies regarding CI/CD servers. This is due that is reliable and open source, which allows companies to save money using a great tool. One of the main drawbacks is that it can be tricky to configure for beginners and the UI is not very user friendly, but once the user has learned how to use Jenkins, the tool allows developers to build complex pipelines.

1.6.2. Source code repositories

There are three source code repositories among the most used by companies: Github (2), Gitlab (3) and Bitbucket (4). All of them can be hosted in the cloud or self-hosted. Github is the most used for opensource projects, Bitbucket provides more integration with other Atlassian products like Jira (5) and Gitlab is the cheapest of the set. They have similar functionalities, and the decision relies in which environment does the company use, what kind of project are they working on and the budget.

1.6.3. Automated analysis tools

The OWASP has recollected a good number of available tools that fall under this technology (6). The tools can be categorized in 4 groups: SAST, DAST, IAST and SCA. There are a lot of tools for each category, and they differ in programming languages supported, price and other features. We will explore more about those groups in the next chapter.

1.6.4. Integration

In order to make the most of the previous tools is necessary to integrate them so the process can be almost automatic. The idea is to create pipelines with Jenkins as the main CI/CD server and configure a code repository and the automated analysis tools to work together.

There are plugins for Jenkins that allow the communications with the repositories mentioned above. For some automated analysis tools there are also plugin to communicate with Jenkins. Some of those tools can be executed as a standalone application, so they can be executed directly by Jenkins.

An interesting use case of the integration is to check for vulnerabilities once a pull request is opened in the code repository. The server will trigger a build in Jenkins and the analysis tools will be executed. Once the scan is finished, if the tools find vulnerabilities in the code, they will notify the code repository server and the

merge pull request will be declined, thus preventing vulnerable code in our application.

1.7 Resources & cost

For this project we will study a variety of the tools that are available, but for the integration in the software development cycle we are going to use only those which are open source or free to use in order to reduce costs.

Resource	Cost
PC	600€
Internet connection	30€/month

1.8. Risks

In this section we will study the risks that could affect the development of this study.

Here are some of the problems that we might encounter during the development of this project:

- **Risk #1 - Scope too broad (High)**: We want to study 4 categories of automated code analysis in a period of 16 weeks. In order to fulfill the project, we have to narrow the samples of the solutions to study.

- **Risk #2 - Difficulties during the integrations (Medium):** the integration of the products will require coding, which might produce errors. If these errors are not solved quickly the project can suffer a delay in the delivery. In those cases, we should focus on completing the main objective of the project, reaching for help from the supervisor or in forums if necessary.

- **Risk #3 - Steep learning curve (Medium):** some technologies might have an initial difficulty of learning very challenging. In those cases, we should search for tutorials and guide steps that allow us to understand that technology.

- **Risk #4 – Illness (Low):** the deadlines are very tight. If I get sick and I can't work the final result is going to suffer a dropdown in quality and quantity. If that happens, we will have to reschedule the planning and prioritize the most important aspects of the project.

1.9. Social and ethic impact

For this project we have not observed any social or ethic impact.

2. Research phase

2.1 Deployment tiers

Before studying the different technologies, it is necessary to enumerate and explain the deployment tiers that are going to be referred to in the next sections. If we plan to integrate these tools in our software development lifecycle, we need to know in which deployment tiers we should integrate them.

For this project we are going to split them into 4 tiers:

1. Development: This tier includes the developer's local workstation and the secondary branches of the source code repository where the developer commits the changes without integrating them into the main branch.

2. Integration: This tier covers the process when the developer opens a pull request, runs unit tests and the changes are merged into the main branch.

3. Preproduction: In this tier we will mirror the final product, where the new features are tested in a real environment before releasing them to production.

4. Production: In this tier is the final product that serves the clients.

In order to automate this deployment process, we will use CI/CD tools, such as Jenkins.

2.2. SAST

In this chapter we will introduce Static Application Security Testing (SAST) technology, how it works, give some examples of current solutions and how they are integrated.

2.2.1 What is SAST and how it works.

Static Application Security Testing (from now on SAST) “is a set of technologies designed to analyze application source code, byte code and binaries for coding and design conditions that are indicative of security vulnerabilities.” (7). It is considered white-box testing, the source code is analyzed from the inside out and the application is not running.

In order to test the source code, SAST tools have a set of predetermined and custom rules. The source code is checked against these rules and the analyzer will evaluate if the code complies with the rules (8). These sets of rules are based on regular expressions and are language dependent.

This technology is not 100% accurate, some of the results might be false positives. In the end it is up to the developer to check if the vulnerabilities are real or not.

There are different types of analysis, each has its own purpose (9):

- Configuration analysis: Check the configuration files to check that there is no bad configuration that may cause security vulnerabilities.

- Semantic analysis: Examines syntax, identifiers and resolving types from code. It is also able to analyze code in its context in search of vulnerabilities.

- Dataflow analysis: Analyze the data flow to see if the data is sanitized or it comes from insecure sources.

- Control flow analysis: checks the order of the program operations to detect vulnerabilities in the sequences of orders.

- Structural analysis: detects language-specific code malpractices, bad design, declaration and use of variables and functions, and hardcoded passwords and cryptographic issues.

2.2.2. SAST solutions

We can find a vast catalogue of tools that provide us with the benefit of static analysis. The Open Web Application Security Project (OWASP) has created a list of the most current known (6).

In order to choose a tool, our criteria should contemplate the following:

- The tool supports the programming language we use
- Commercial or free license
- How can it be integrated into the software development lifecycle

We will take a look at some of them:

HCL AppScan CodeSweep

HCL AppScan CodeSweep (10) is a plugin for VSCode and IntelliJ IDE by HCL Technologies. It scans the code while the developer is working in search of vulnerabilities.

- License: Open Source or Free

- Languages supported: Java, .Net, Go, Python, Ruby, JS PHP, Perl, COBOL, Apex.

CxSAST

CxSAST (11) is the SAST solution by Checkmarx. It is commercialized as SaaS or on-premises. It has CI/CD integration plugins available.

- License: Commercial
- Languages supported: Javascript, Java, Apex, PHP, Python, Swift, Scala, Perl, Groovy, Ruby, C++, C#.NET, PL/SQL, VB.NET, ASP.NET, HTML 5, Go, and Kotlin

SonarQube

SonarQube (12) scans source code for Bugs, Vulnerabilities, and Code Smells. It has plugins to integrate it in CI/CD pipelines and a module to fail a build if the quality of the code is not good enough determined by a set of rules.

- License: Open Source or Free / Commercial
- Languages supported: Java, C#, JavaScript, TypeScript, CloudFormation, Terraform, Kotlin, Ruby, Go, Scala, Flex, Python, PHP, HTML, CSS, XML and VB.NET

Fortify

Fortify (13) is a SAST tool that supports more than 30 languages and frameworks. It has CI/CD integration plugins at its disposal.

- License: Commercial
- Languages supported: ABAP/BSP, ActionScript/MXML (Flex), APEX, ASP.NET, VB.NET, C\#, C/C++, Classic ASP COBOL, ColdFusion CFML, Go, HTML, Java JS/AJAX, JSP, Kotlin, Objective-C, PHP, PL/SQL, Python, Typescript, T-SQL, Ruby, Scala, Swift, VB.NET, Visual Basic 6, VBScript, XML

2.2.3. SAST integration

SAST can be integrated in the development, integration and preproduction tiers.

In the development tier, SAST can be integrated as an extension for the IDE. These tools scan the code and inform the developer about potential issues while the developer is working. Some tools can be integrated in the IDE like HCL AppScan CodeSweep.

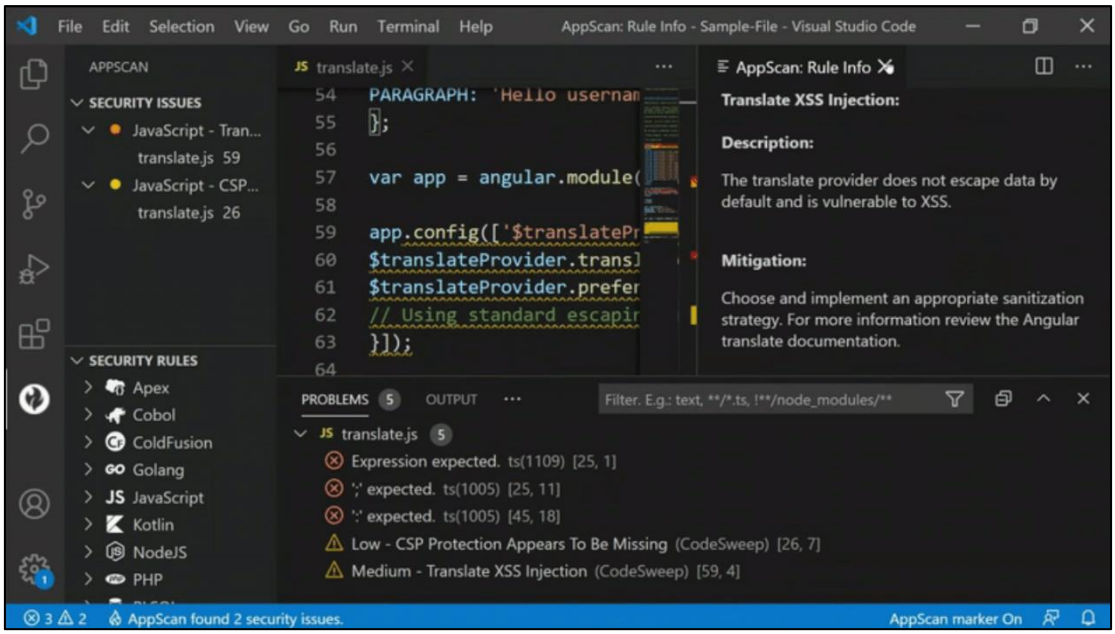

Figure 1 A diagnosis of security issues in Visual Studio Code. Source: HCL AppScan CodeSweep

In the integration and preproduction tier, SAST can be automated in a CI/CD pipeline. For example, Checkmarx and Snyk have plugin that allow Jenkins to run a scan. This, added to the interoperability that Jenkins has with source code repositories like Stash, a scan can be executed after opening a pull request or while running a build for preproduction

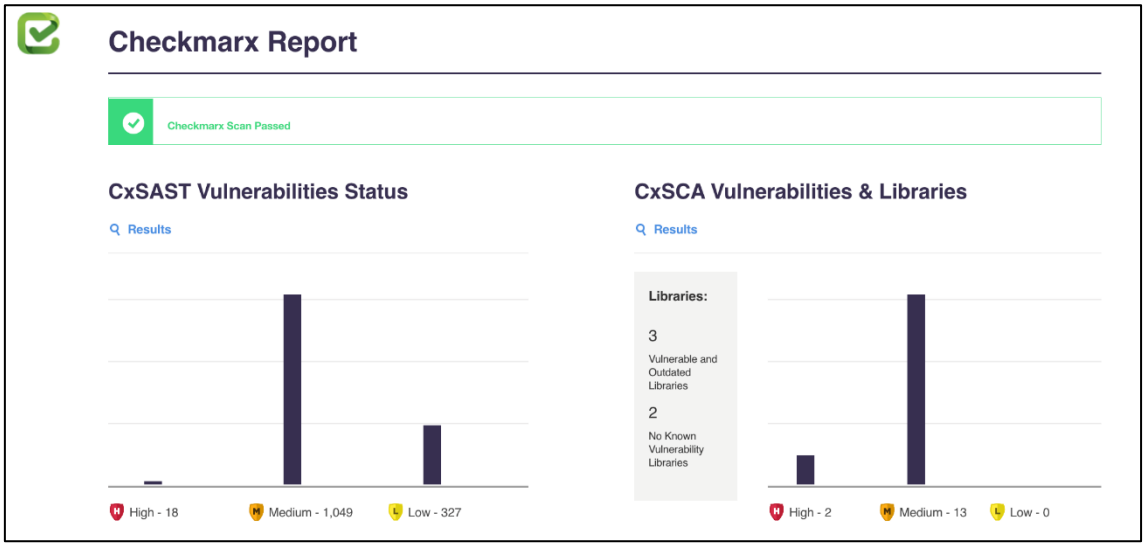

Figure 2 Jenkins's Checkmarx plugin report after a build. Source: Checkmarx.com

There is also the possibility to run a scan manually at any time, but we are focused on automating this process.

2.3. DAST

In this chapter we will introduce Dynamic Application Security Testing (DAST) technology, how it works, give some examples of current solutions and how they are integrated.

2.3.1 What is DAST and how it works.

Dynamic Application Security Testing (from now on DAST) is “the process of analyzing an application to find vulnerabilities through simulated attacks. This type of approach evaluates the application from the “outside in” by attacking an application like a malicious user would.” (14).

Unlike SAST, DAST is black-box testing. The tools don't know how the application works and cannot see the code. The tools use a set of tests that are language-independent and mimic the attacks of malicious hackers to discover vulnerabilities in our software. These tests can be predefined or written by experts and are performed while our application is running. Then, the solution analyzes the responses from the server after malicious requests are performed. However, like SAST, it can also return false positives that the developer must go through to verify if the threat is real or not.

But the main problem of using DAST solutions in a pipeline is that in order to test as many attacks as it can, it can take a huge amount of time. We will discuss how to integrate DAST in the next sections.

2.3.2. DAST Solutions

For DAST we can also find a lot of tools available in the market. There are updated lists of DAST tools that we can find on the internet (15).

In this case, since DAST works as a black box, we don’t have to worry about the language we have implemented our application. Let see some examples:

OWASP ZAP

OWASP ZAP (16) is the OWASP DAST tool. It is a free and open source web scanner. It can be used both manually or automatically. It is supported by the community and have a lot of plugins available with different test suites for specific vulnerabilities.

- License**:** Open source or free

Nikto

Nikto (15) is a web server scanner that searches for over 6000 vulnerabilities, including configuration files. It detects version specific problems.

- License**:** Open source or free

Acunetix

Acunetix (17) is a web scanner made by Invicti. It has a desktop and a cloud version. It crawls a website in order to find vulnerabilities. It also provides an inventory of the existing assests.

- License**:** Commercial

InsightAppSec

InsightAppSec (18) was created by Rapid7, this web scanner tests for more than 95 kinds of attacks, has a low false positive rate and cover the top 10 OWASP vulnerabilities.

- License**:** Commercial

2.3.3. DAST Integration

Since DAST works on a running application, it can be integrated in the preproduction and production tiers.

As we mentioned before, the main problem integrating DAST is the time it consumes. A thorough test can last 5-7 days (19). For a building CI/CD pipeline this is too much, so what companies do is create a separate job to run DAST on a weekly basis or before a release.

The ideal scenario is that preproduction mirrors exactly what is already or is going to be in production, so DAST can be performed in preproduction and avoid unnecessary traffic in production.

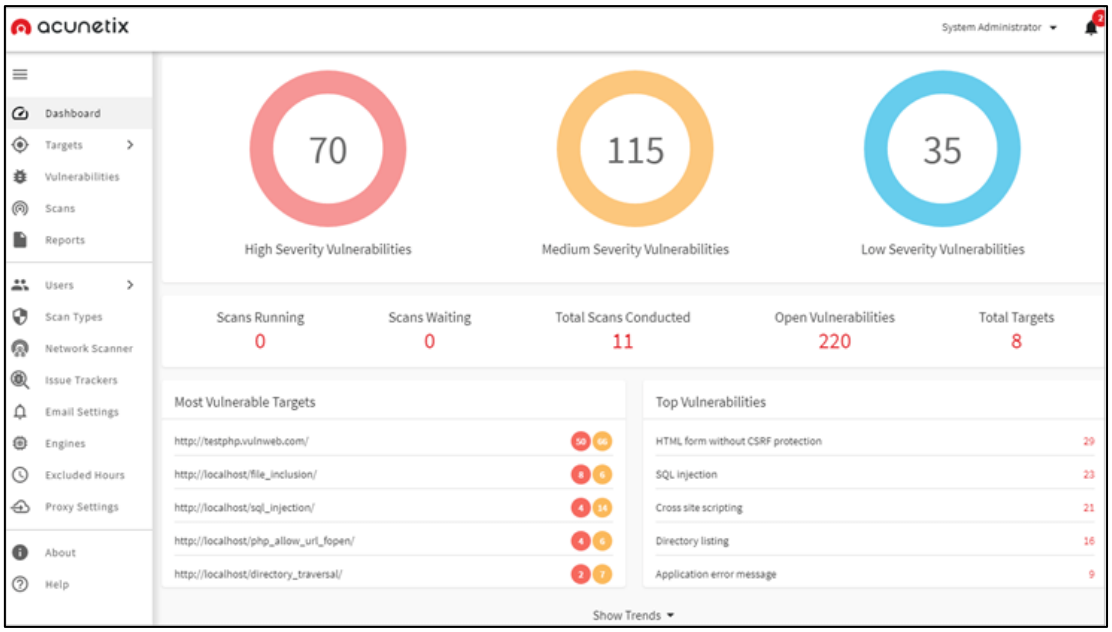

Figure 3 Acunetix DAST report. Source: https://www.softwaretestinghelp.com/

2.4. IAST

In this chapter we will introduce Interactive Application Security Testing (IAST) technology, how it works, give some examples of current solutions and how they are integrated.

2.4.1 What is IAST and how it works.

Interactive Application Security Testing (from now on IAST) is an “application security tool that focuses on the detection of security issues in the code of your applications. Designed to run in the application server as an agent, they provide real-time detection of security issues by analyzing the traffic and the execution flow of your applications” (20).

IAST uses instrumentation, which means an agent injects functionality in the app code. This way, when a vulnerability or error is found, the exact line of code that produces it can be pinpointed. Like DAST, it is used on a running application.

There are two types of IAST:

- Active IAST (Partial IAST): active IAST (or partial IAST) combines a web scanner or inducer (such as DAST) and an agent or detector. The inducer attacks the application with a suit of tests and in case a vulnerability is found, the detector pinpoints the code where it happens.

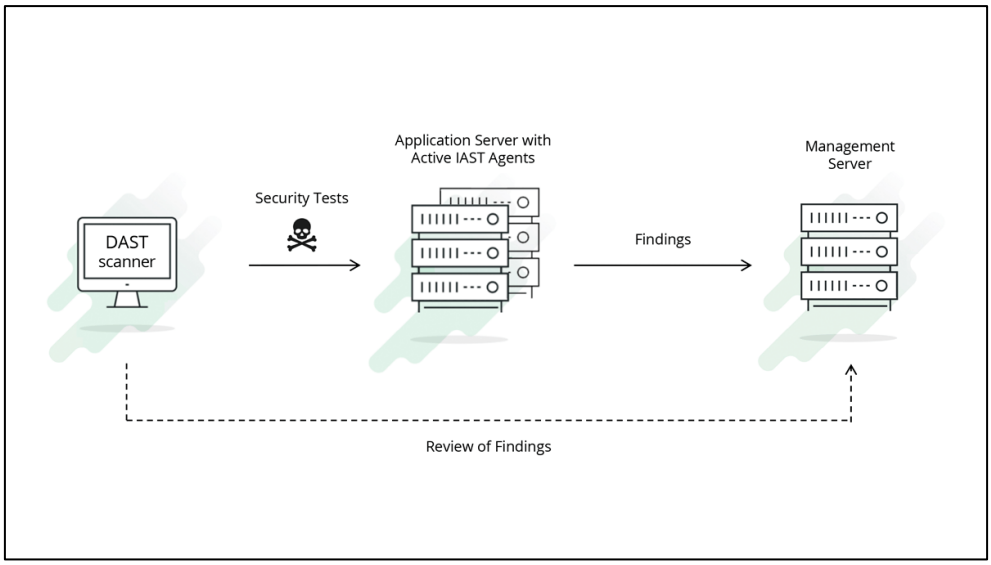

Figure 4 Active IAST flow. Source: hdivsecurity.com

- Passive IAST (Full IAST): Passive IAST (or full IAST) only uses the detector component as a runtime agent in the server. It uses all the traffic (regular users, quality testers, …) to find errors and vulnerabilities in the code.

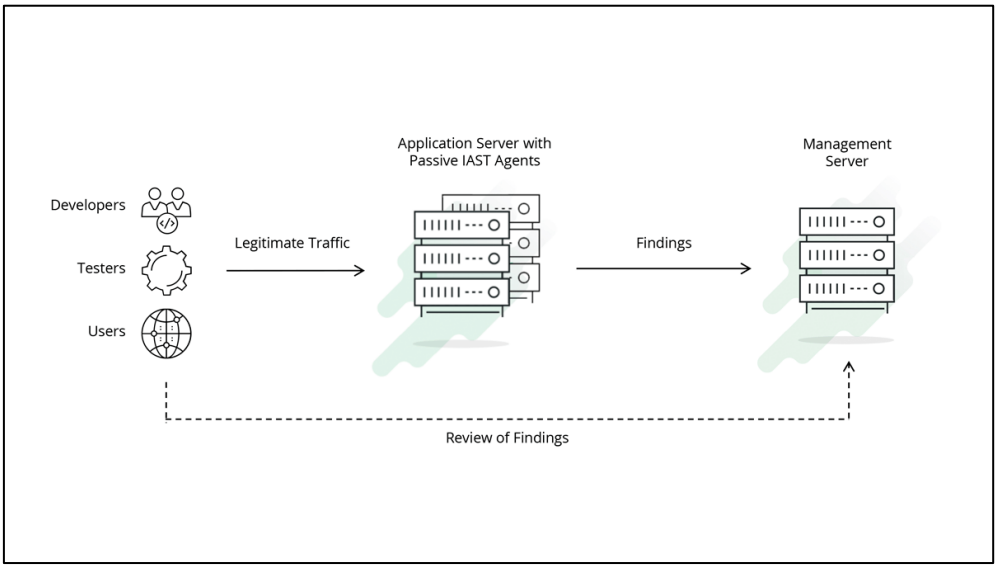

Figure 5 Passive IAST flow. Source: hdivsecurity.com

Active IAST has some drawbacks. One of them is the time it takes to complete since it uses DAST to locate the vulnerabilities, and as we saw in the previous section that it can last days until it is finished. The other main drawback is that it will only analyze the code that is affected by DAST's test suite, so the code that is not affected by DAST request could have potential vulnerabilities.

2.4.2. IAST Solutions

There is not any open source solution for IAST. During the research only one product was found that have a free plan. Like SAST, IAST is language dependent.

Acusensor

Acusensor (21) is the IAST module for Acunetix. It supports Node.js, PHP, Java and ASP.NET. It provides a combination of DAST + IAST in one tool to make the integration between these tools easier.

License: Commercial

Contrast

Contrast (22) is the only IAST tool that has a community edition that is free to use. While the commercial version supports more than 12 languages, the community edition only supports java, node.js and .NET.

- License: Commercial/Freemium

CxIAST

CxIAST (23) is the IAST solution by Checkmarx. If used with the rest of Checkmarx products provides a great flow from integration tier to production tier with integration plugins for CI/CD.pipelines.

- License: Commercial

2.4.3. IAST Integration

Like DAST, IAST works on a running application, so it can be integrated in the preproduction and production tiers.

The best approach is to implement passive IAST in preproduction and production, so it will catch errors and vulnerabilities as soon as the developers start testing the application and will continue to do it when the final users interact with the application.

If DAST tools are integrated, IAST will help to locate the code where the vulnerability is generated, so the best practice is to implement IAST before running the job that will execute DAST.

2.5. SCA

In this chapter we will introduce Software Composition Analysis (SCA) technology, how it works, give some examples of current solutions and how they are integrated.

2.5.1 What is SCA and how it works.

Software composition analysis (from now on SCA) is “an automated process that identifies the open source software in a codebase. This analysis is performed to evaluate security, license compliance, and code quality.” (24)

This technology analyzes manifest files, containers, source code and binaries among other resources to elaborate a Software bill of materials (SBOM) that is checked against public and private databases to get information about vulnerabilities and licenses. Depending on the analyzed resource, the tool will scan it accordingly: for manifest files will query the packet manager to extract the information, but for binaries and sources will calculate the hash of the resource or search the code for identifiers to match them against a database.

2.5.2. SCA Solutions

Like the other technologies, there are a lot of solutions available in the market. For these products we will look at their license, some of their features and how to integrate them in our pipeline.

CxSCA

CxSCA (25) is the SCA solution by Checkmarx. It can scan library files and package managers manifests. It also detects the licenses of third-party libraries and send alarms in case the licensing breaks the company policies. It has Jenkins plugins to integrate it in the CI/CD pipeline.

- License**:** Commercial

Retire.js

Retire.js (26) is an open source solution for javascript SCA. It runs as a command line program and scans the library files to search vulnerabilities. It includes multiple output formats, including SBOM. It has a database with the most used libraries. It doesn’t have Jenkins plugins to integrate it.

- License**:** Open source/free

Mend SCA

Mend SCA (27) is the SCA solution by Mend. It has a wide variety of libraries and supports multiple languages. One key feature is that it can automatically open a pull request to update the libraries that are outdated or have vulnerabilities. Like Checkmarx, it has licensing compliance and CI/CD plugin integration.

- License**:** Commercial

2.5.3. SCA Integration

SCA scans are used in the integration and preproduction tier. SCA tools integration is similar to SAST. These tools can run dependencies scans as a stand-alone, but they are usually integrated in the CI/CD using plugins. If the tool has also SAST capabilities, the plugin often let the user run both scans. (28)

2.6. Research conclusions

After investigating the technologies, we can decide which and how are we going to implement them in the software development lifecycle.

We have decided to use tools that are open source or have a free planning, so every company or developer could implement the following environment in its software development lifecycle, regardless of their budget.

2.6.1. Topology in development tier

In this tier we decided to use HCL AppScan CodeSweep, a SAST tool that works as live code scanner. It is distributed as a plugin for code editors. When the developer is programming, the code editor triggers HCL AppScan CodeSweep scan. Then, it returns the security issues to the code editor that displays them to the developer.

In our environment we are going to use Visual Studio Code (29),a lightweight code editor by Microsoft that has plenty of plugins at its disposal.

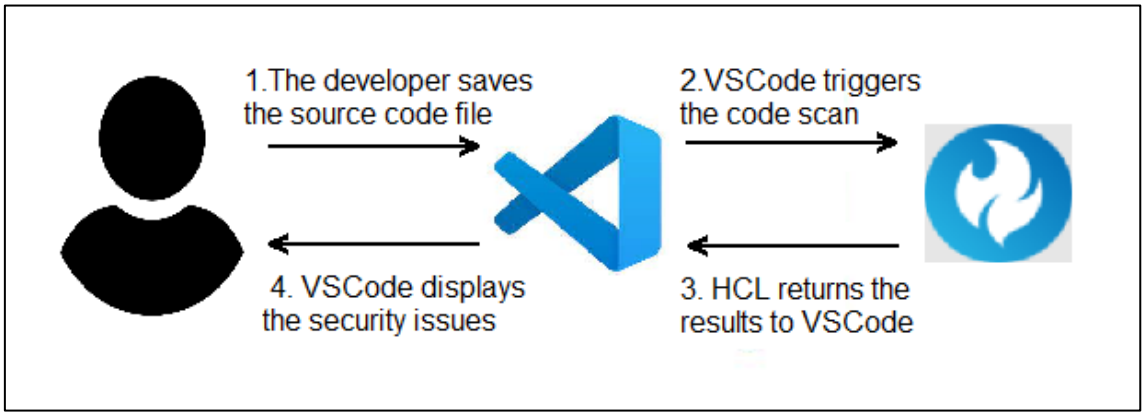

Figure 6 Development tier topology

2.6.2. Topology in integration tier

In the integration tier we are going to implement Sonarqube (12) and retire.js (26) in our environment.

Sonarqube is a SAST tool that will scan the source code for vulnerabilities and will evaluate the code quality regarding maintainability, code repetition and unit tests coverage. The scan results are stored in the SonarQube server.

Retire.js is a SCA tool that has a database (26) with known security issues of the most popular third-party libraries and analyzes the libraries of the project against that database. It will return the results in a JSON file.

These tools are going to be implemented in a Jenkins (30) pipeline. Jenkins is an open source automation server for continuous integration and development. It is used to build, test and deploy applications.

Also, we are going to use Bitbucket cloud (4), a cloud source code repository in which our sample application code is going to be stored.

When a developer creates a pull request to merge a branch in the code repository, a build will be triggered in Jenkins. Jenkins will proceed to retrieve the code, build the project and run both tools. If the scans results are not good enough for our standards the job will fail, and the developer won't be able to merge the branch until the job is successful.

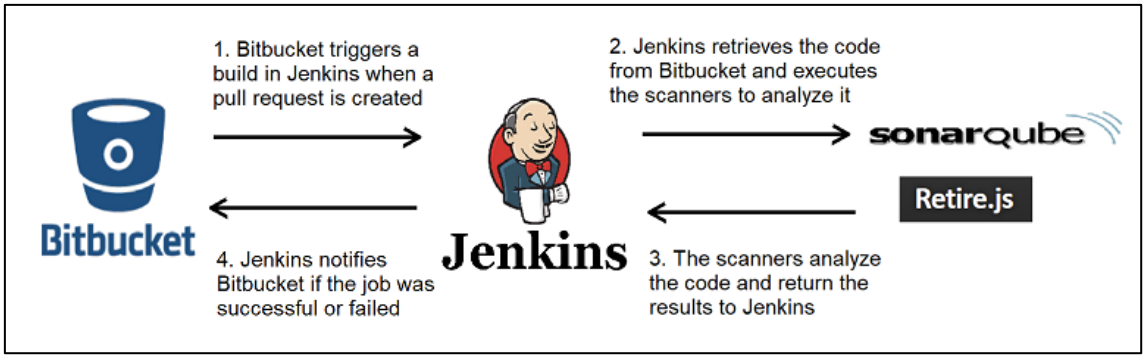

Figure 7 Integration tier topology

2.6.2. Topology in preproduction and production tiers

In these tiers we are going to use OWASP ZAP (16) and Contrast IAST (22).

Contrast has a free planning for IAST. It can install an agent in a Java, Node.js or .NET application and inspect all the incoming traffic in search of vulnerabilities. The results can be consulted in Contrast server.

OWASP ZAP is a DAST tool that contains tests for the most common vulnerabilities. It has different modes to be executed, but for our integration we are going to use the command line mode.

In our pipeline, Jenkins is going to execute OWASP ZAP against a running instance of a web application. This web application is already instrumentalized with Contrast IAST, so the request made by OWASP ZAP will be analyzed by IAST technology too. Finally, when DAST tests are finished, OWASP ZAP returns the results to Jenkins in a XML file.

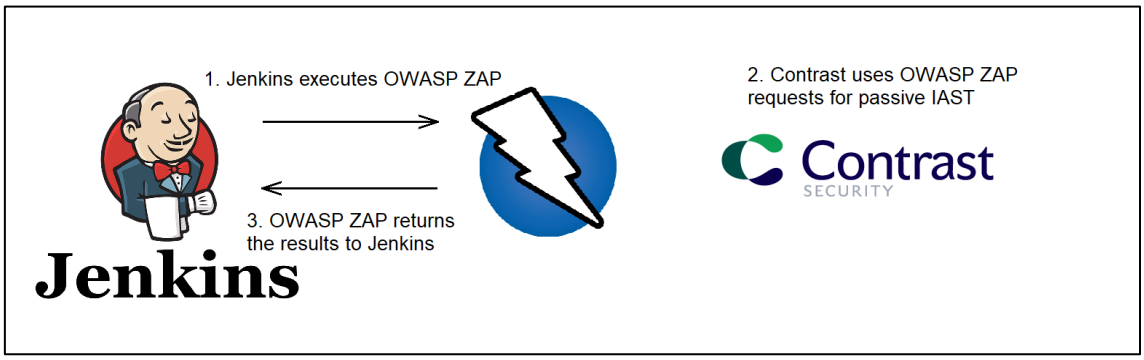

Figure 8 Preproduction and production tier topology

3. Implementation phase

3.1. Environment

In this section we will enumerate all the technologies that are going to be used to implement the different solutions to analyze our code. The environment is going to be implemented mostly in a local host, but the same configuration can be applied for a cloud architecture.

3.1.1. Technology

Here is a brief introduction of the software on which we will build the environment:

- **Ubuntu** (31)**:** Ubuntu is a Linux operating system. It is the OS in which all the applications are going to be executed.

- **JDK** (32)**:** The java development kit. It includes the tools to compile and run java code. It is necessary to run some tools (e.g., Jenkins) that we are going to use in the environment.

- **Node.js** (33)**:** Node.js is a javascript framework for creating web applications in a fast and easy way. We are going to use it to create a sample application that will be scanned by the different solutions in our continuous integration pipeline.

- **Docker** (34)**:** A containerization software that allows us to run containers, that is, standalone packages of software that include everything that is necessary to run an application.

- **Ngrok** (35)**:** It is a software that allows us to expose a local machine and port to the rest of the world through a custom domain in an easy way. We are going to use it to expose our Jenkins instance and make it accessible to Bitbucket cloud.

- **NPM (Node Package Manager)** (36)**:** NPM is a package manager for the JavaScript programming language. We are going to use it to download Retire.js.

- **jq** (37): is a lightweight and flexible command-line JSON processor, very useful for working with JSON results.

3.2. Implementation of the automated code analysis tools

In this section we will implement the different solutions that will analyze our application through all the deployment tiers. All the solutions are open source or

have a free version of the product, so every company or developer could implement the following environment in its software development lifecycle, regardless of their budget. The only exception is Bitbucket cloud, which charges for teams that have more than 5 users.

3.2.1. Implementation in the development tier

In this tier we are going to implement HCL AppScan CodeSweep. This SAST tool scan the source code while the developers are programming. It is available as a plugin for Visual Studio Code and Intellij IDEA (38).

In order to install it, we search it on the extensions tab in Visual Studio Code and click install.

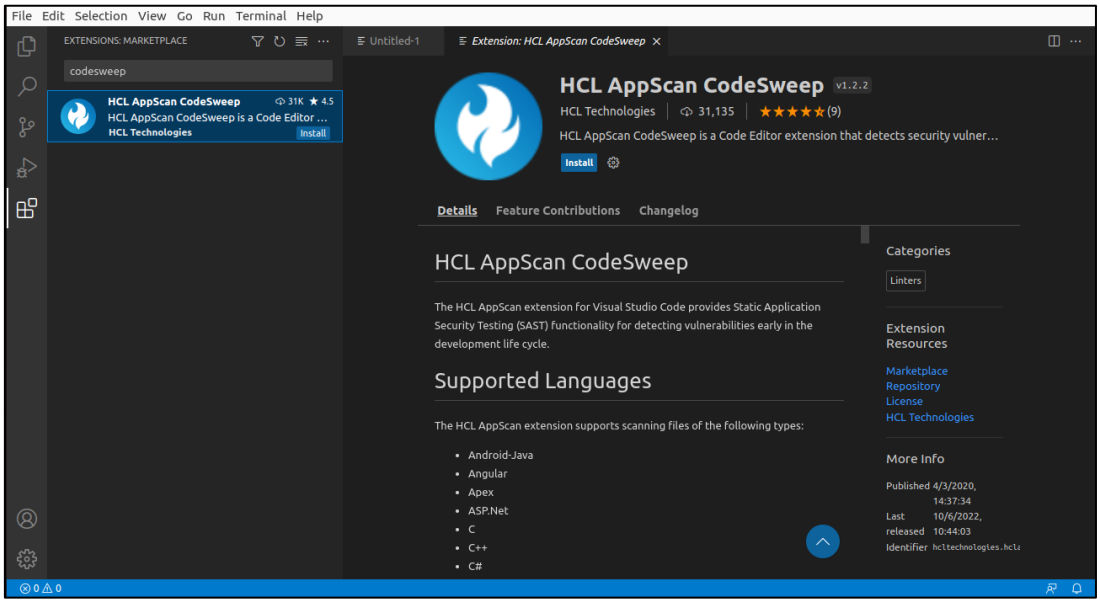

Figure 9 HCL AppScan CodeSweep in VSCode Marketplace

Once the extension is installed, a new icon will appear on the left bar. If we click it, we will open the tool panels. On the left panel we can find a list of security rules categorized by programming languages/technologies. If we expand a category, we can find all the rules that are checked for a certain technology. We can learn more about the rules by clicking on the information icon next to the rule. This will open a panel on the right with related information about the rule.

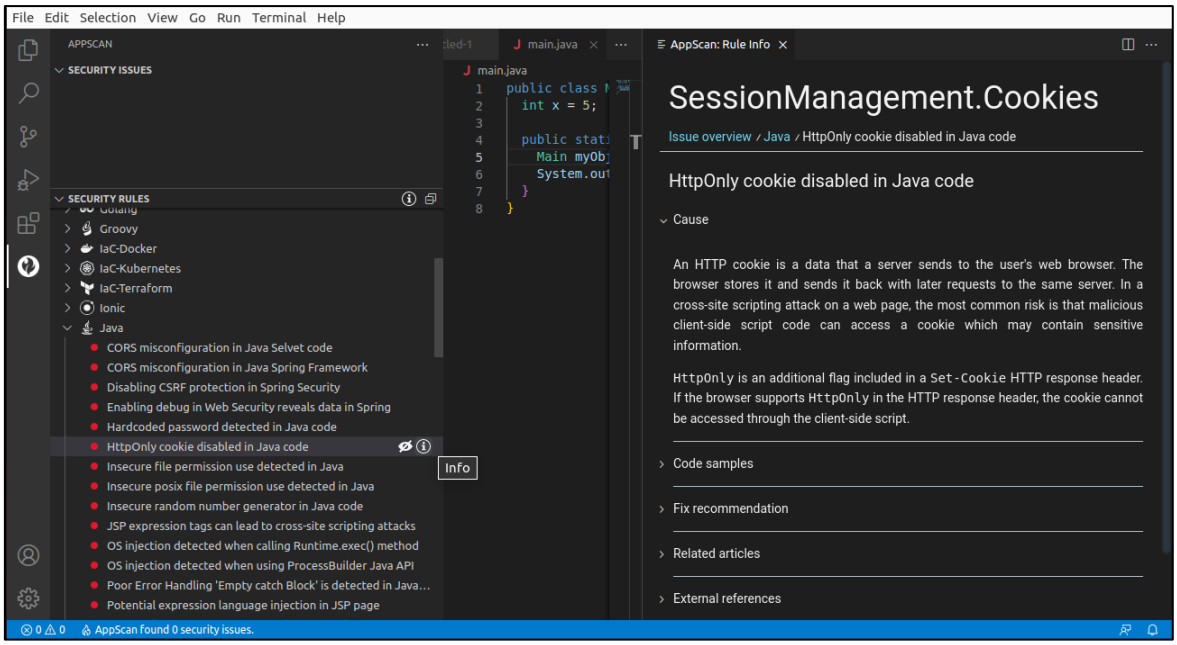

Figure 10 HCL AppScan CodeSweep avaliable rules

Now that the tool is installed, if the developer's code has some vulnerabilities that are matched against the security rules, the problematic code will be underlined, and the problem will be shown in the "problems" tab and in the "security issues" panel.

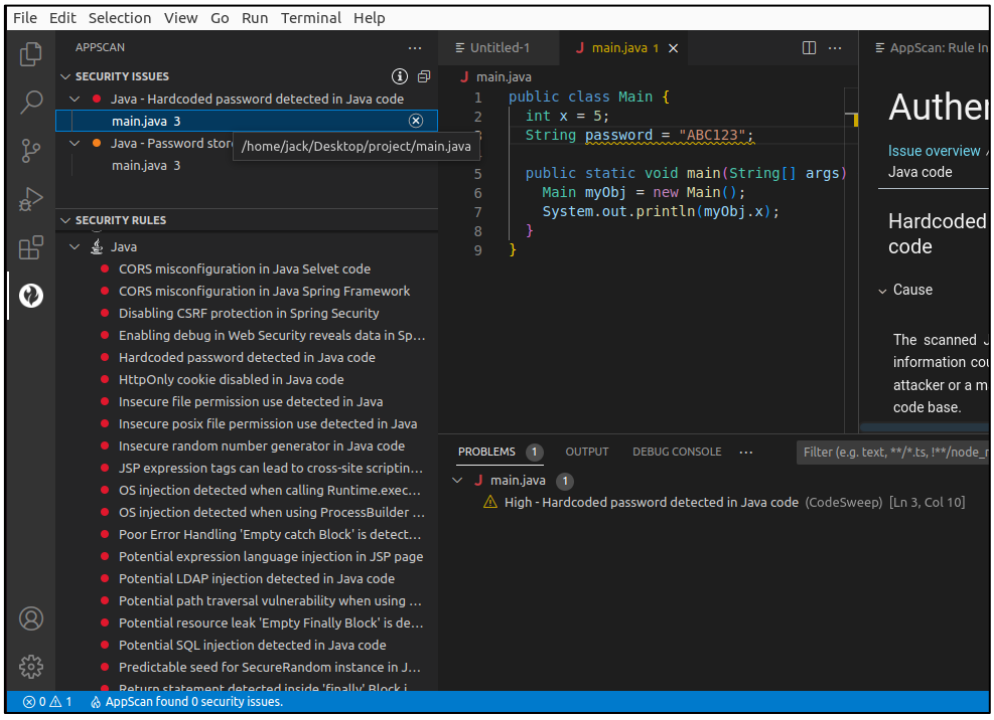

Figure 11 Vulnerabilities found by HCL AppScan CodeSweep

If we inspect the issue, we will find more information on the right panel. There we can learn about what are the security problems, see examples, measures to fix them and related articles and references for further reading.

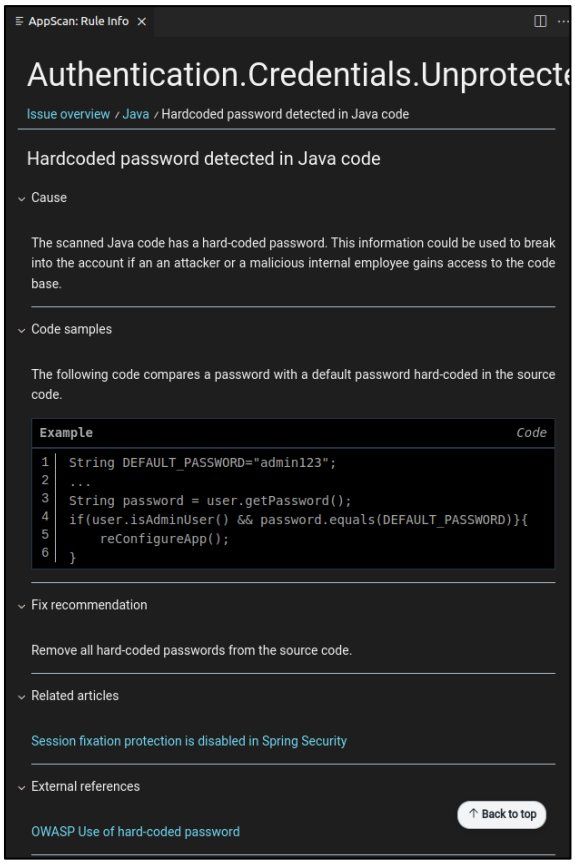

Figure 12 Information about a vulnerability in HCL AppScan CodeSweep

3.2.2. Implementation in the integration tier

In this section we are going to see how to integrate SAST and SCA tools in our environment. First, we have to set up the code repository and Jenkins, so we can install SonarQube and Retire.js later.

3.2.2.1. Setting up Bitbucket Cloud and Jenkins

In order to create the architecture explained in the previous section, it is necessary to configure Bitbucket Cloud and Jenkins so they can communicate with each other.

Since Jenkins is running in localhost it is necessary to expose it to the internet so Bitbucket Cloud can reach it. We will use Ngrok for this purpose: with an account already created we only have to run the command `ngrok http 8080` (the port in which the Jenkins instance is running).

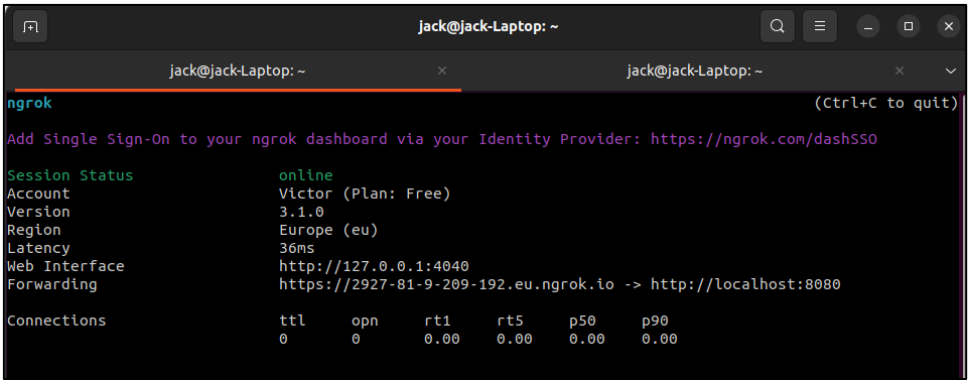

Figure 13 Running Ngrok to expose the Jenkins instance

Now that our Jenkins instance can be reached, let's configure Bitbucket Cloud. In our project repository settings, first we will add restrictions to merge into the main branch. We navigate to "Branch restrictions", select the development branch, and in "Merge settings" we activate the checkbox "Minimum number of successful builds for the last commit with no failed builds and no in progress builds" and set it to 1. This way only the branches that have a successful build can be merged into the main branch.

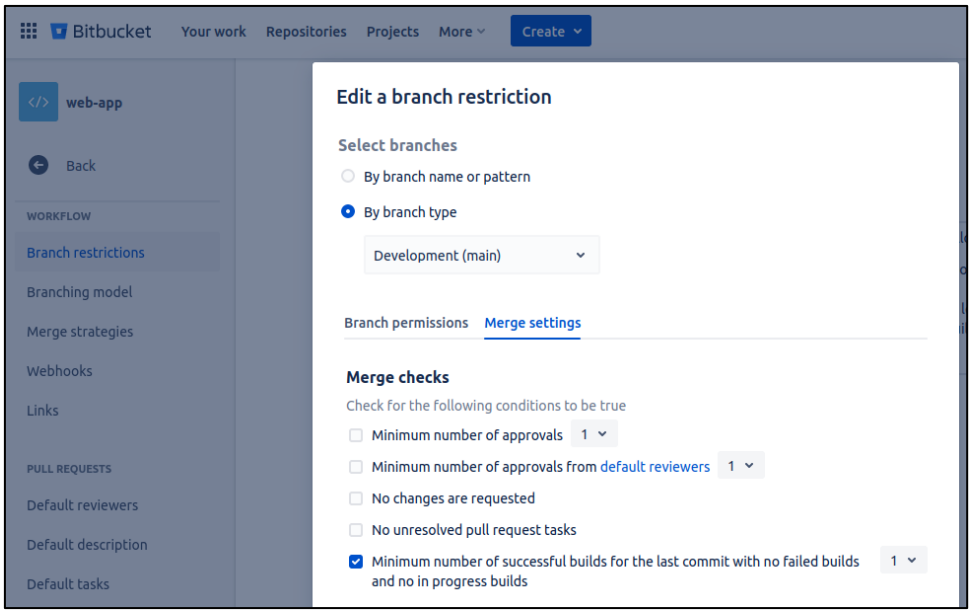

Figure 14 Bitbucket branch restrictions

Now we must create a webhook so it can communicate with our Jenkins instance. A webhook is an HTTP-based callback that expands the behavior of a web page and allows the interoperability of two web applications. In the repository setting we navigate to "webhooks", introduce the URL of our Jenkins instance followed by */bitbucket-scmsource-hook/notify* (this is due to the Jenkins plugin that we are going to use later), and activate the checkbox that triggers the webhook when a pull request is created or upgraded. With this configuration, when a pull request is created or updated, bitbucket will trigger a build in Jenkins.

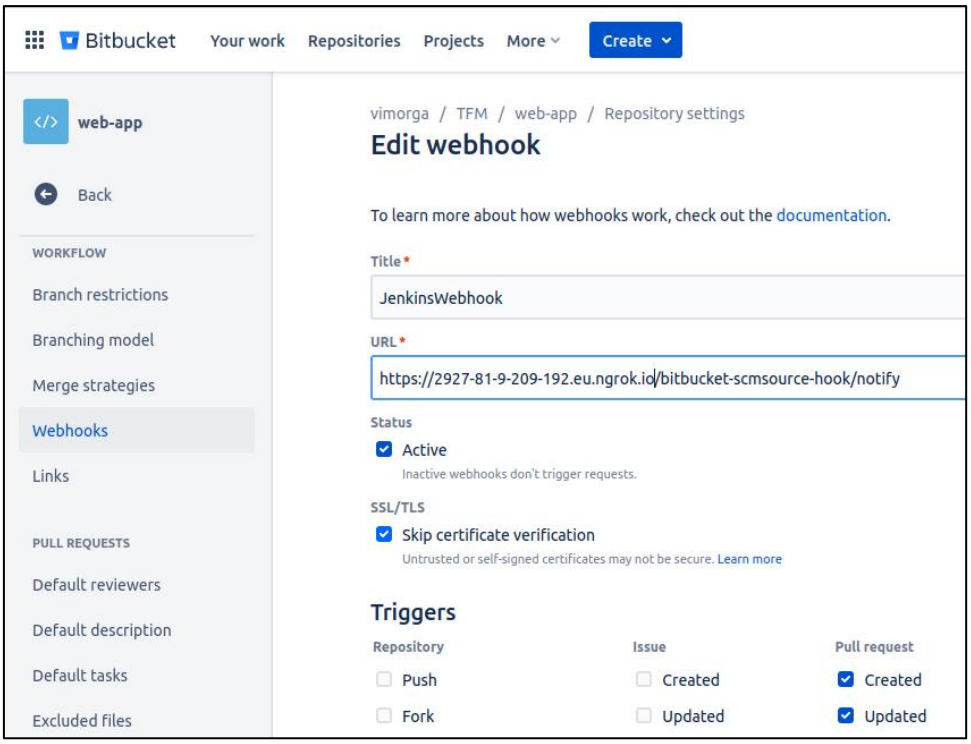

Figure 15 Bitbucket webhook configuration

The last step to configure Bitbucket is to create an OAuth consumer for Jenkins in the workspace settings. Enter a name, the URL of Jenkins and the permission to write the repository and save.

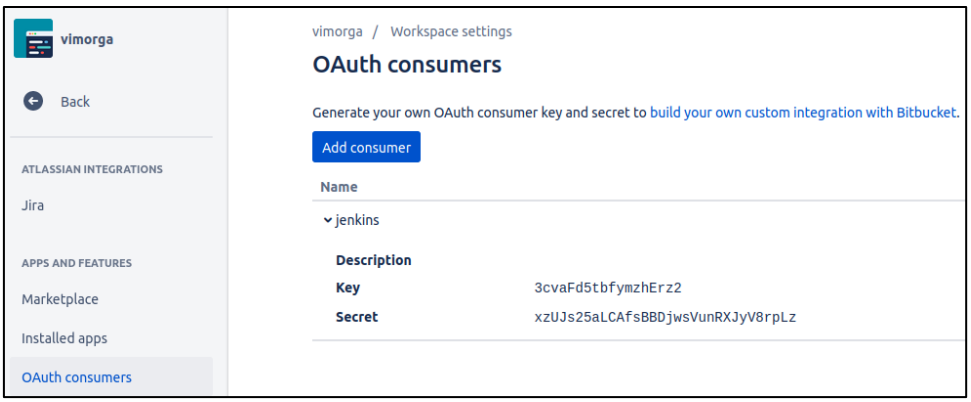

Figure 16 Bitbucket OAuth consumer configuration

Now let's configure Jenkins. The first step is to install the necessary plugins to communicate with Bitbucket:

- Bitbucket Branch Source Plugin (39): allows to use Bitbucket Cloud and Bitbucket Server as sources for multi-branch projects

- Bitbucket Build Status Notifier Plugin (40): is a Bitbucket build status notifier that can publish your build status to Bitbucket Cloud

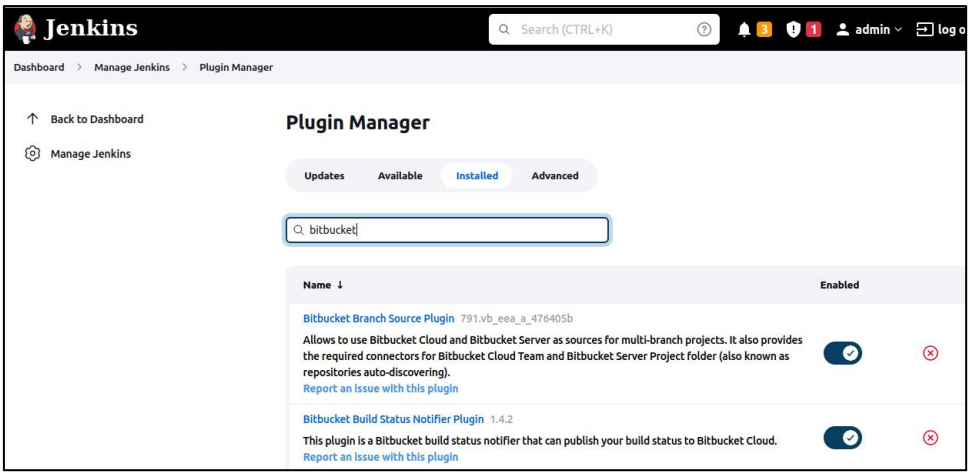

Figure 17 Jenkins plugins for Bitbucket

To finish the configuration, we have to navigate to “Configure System” and add the OAuth credentials in the Bitbucket Build Status Notifier Plugin Section.

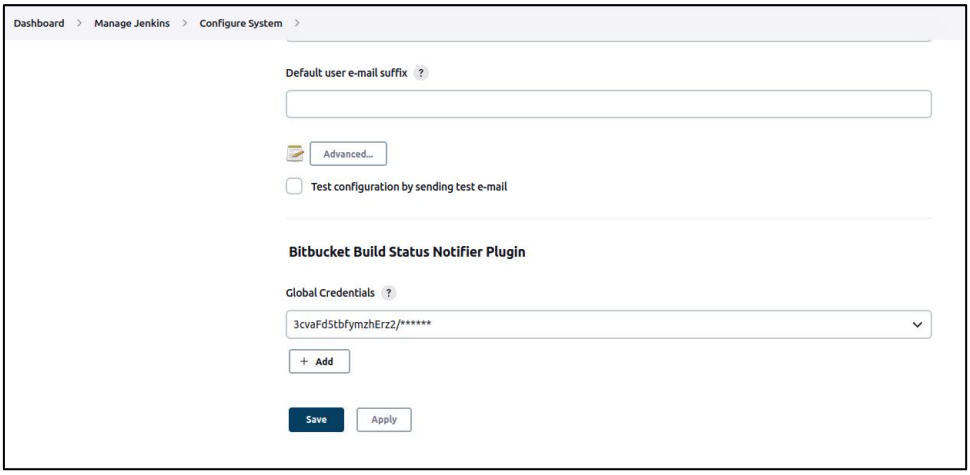

Figure 18 Bitbucket Build Status Notifier Plugin configuration

3.2.2.2 Sonarqube set up

For the Sonarqube integration we are going to use a Docker container as the server. This architecture can be used in a cloud environment like AWS EC2 (41) or Azure VMs (42).

The first step is to download the official Docker image of Sonarqube.

```
jack@jack-Laptop:~$ sudo docker pull sonarqube
Using default tag: latest
latest: Pulling from library/sonarqube
9621f1afde84: Pull complete
4c884cb0d3d1: Pull complete
cb4d01bc5fb2: Pull complete
Digest: sha256:d01fc01edd48c0fcdd8841255cfc30eb05b43e160b4c1b9056ca0c75d32ac285
Status: Downloaded newer image for sonarqube:latest
docker.io/library/sonarqube:latest
```

Figure 19 Download of SonarQube docker image

Then we launch the container with the command "`sudo docker run -d --name sonarqubeserver -p 9000:9000 sonarqube`"

Once the container is running, we can access SonarQube server UI at localhost:9000. First, we must create a webhook so SonarQube can communicate with Jenkins. We navigate to Administration > Configuration > Webhooks and fill the form with our Jenkins instance URL followed by */sonarqube-webhook/* (This is due to the Jenkins plugin that we are going to install later)

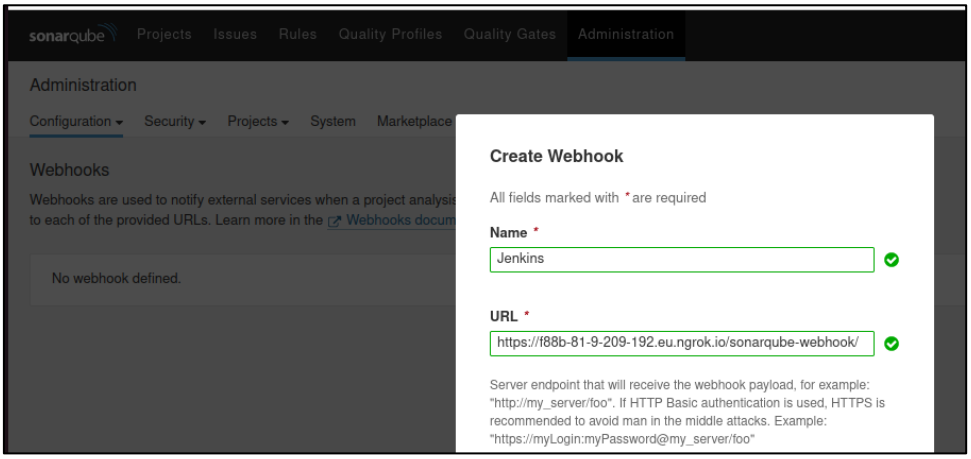

Figure 20 SonarQube webhook configuration

Now we have to create a token for Jenkins to access SonarQube. We navigate to My Account > Security and generate a new Token

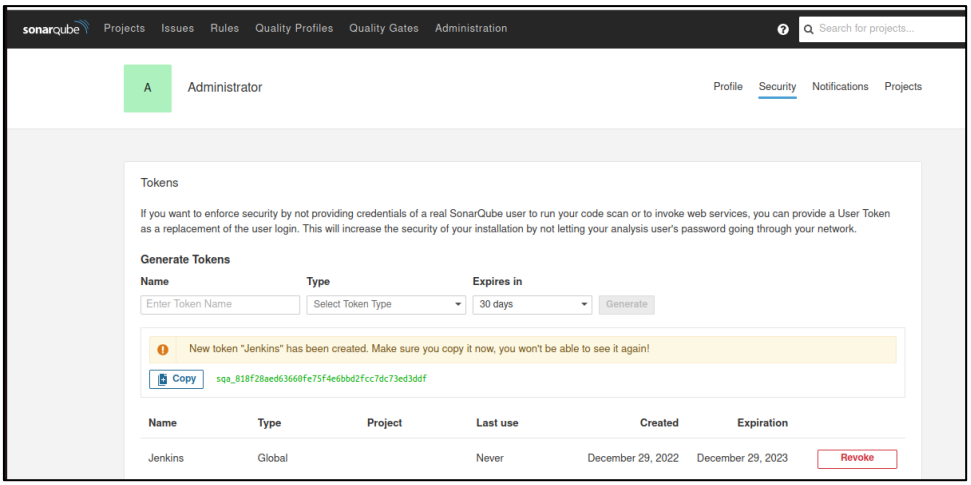

Figure 21 SonarQube Access token generation

In Jenkins we have to install the SonarQube Scanner for Jenkins plugin (43) so it can work along SonarQube server.

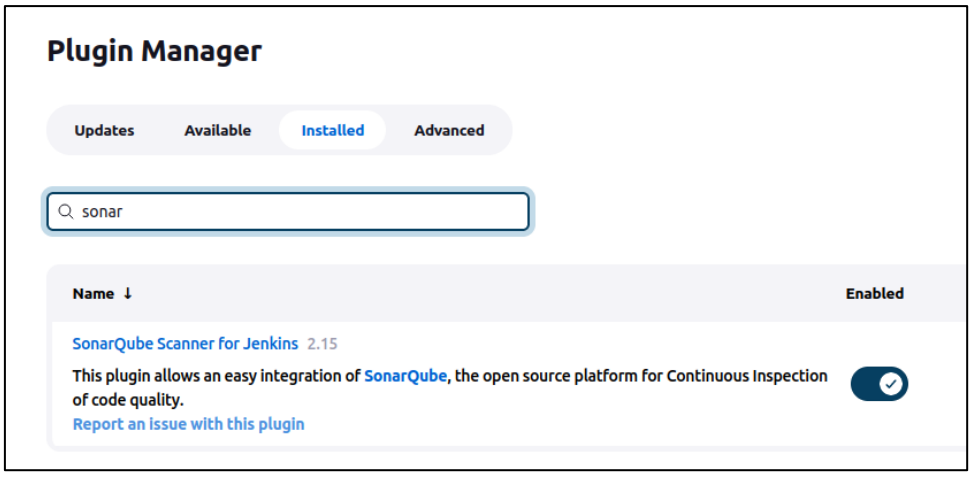

Figure 22 SonarQube plugin for Jenkins

Once the plugin is installed, we configure it in Manage Jenkins>Configure System. We enter the URL and the SonarQube token created in the previous steps

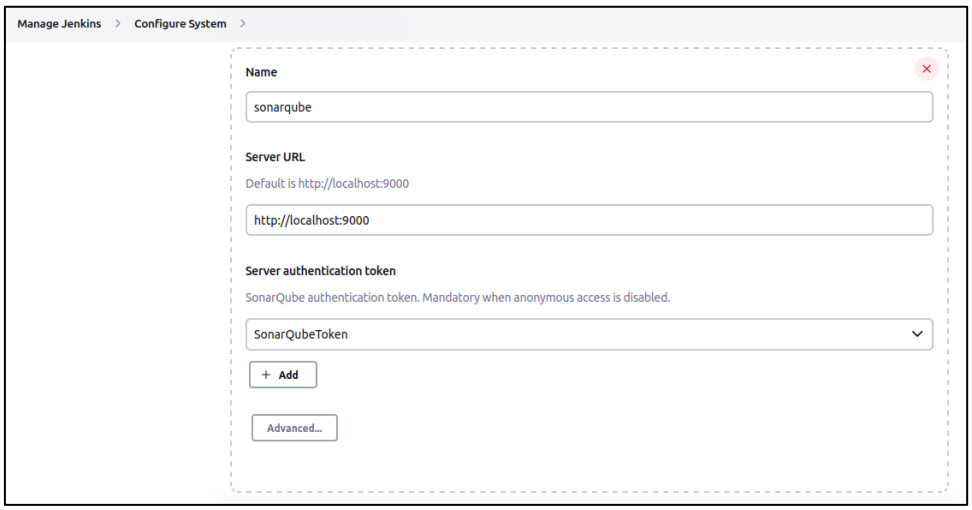

Figure 23 SonarQube plugin configuration

SonarQube has a module called Quality Gates (44). This module allows the developers to establish conditions to determine if a build fails or is successful depending on certain parameters. These parameters are unit tests coverage, percentage of duplicated lines, maintainability rating, reliability rating, security hotspots and overall security rating. With Quality Gates we can set a minimum standard for pull requests to be merged.

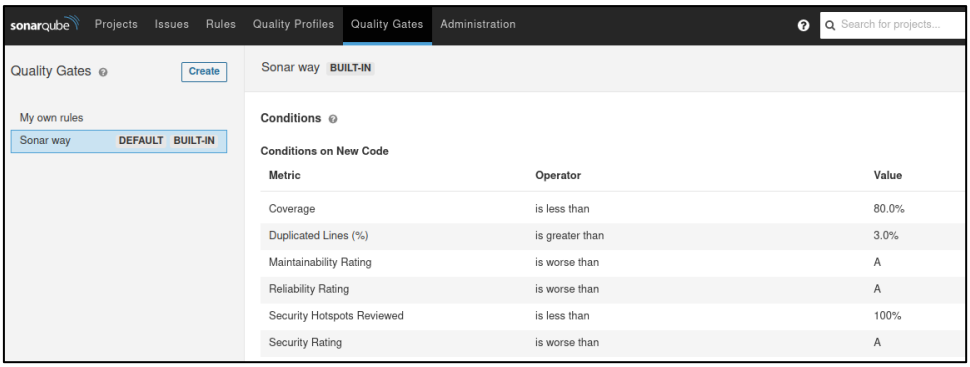

Figure 24 Quality Gates conditions

3.2.2.3 Retire.js set up

We are going to install Retire.js in the machine where Jenkins is installed (in our case is localhost). To do this, we can install it with NPM:

```
jack@jack-Laptop:~$ sudo npm install -g retire
[sudo] password for jack:
npm WARN deprecated readdir-scoped-modules@1.1.0: This functionality has been moved to @npmcli/fs

added 99 packages, and audited 100 packages in 14s

4 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 25 Installation of Retire.js

Retire.js can be executed from the terminal and it will search for vulnerabilities of third-party libraries in a specified directory.

```
jack@jack-Laptop:~/Desktop/project$ sudo retire --outputformat json | jq
DEPRECATION NOTICE: The node scanning is deprecated and will be removed soon. See http
ode.js-scanner
{
  "version": "3.2.1",
  "start": "2022-12-29T21:24:06.838Z",
  "data": [
    {
      "file": "/home/jack/Desktop/project/jquery-3.5.0/test/data/jquery-1.9.1.js",
      "results": [
        {
          "version": "1.9.1",
          "component": "jquery",
          "detection": "filename",
          "vulnerabilities": [
            {
              "info": [
                "https://github.com/jquery/jquery/issues/2432",
                "http://blog.jquery.com/2016/01/08/jquery-2-2-and-1-12-released/",
                "https://nvd.nist.gov/vuln/detail/CVE-2015-9251",
                "http://research.insecurelabs.org/jquery/test/"
              ],
              "below": "1.12.0",
              "atOrAbove": "1.4.0",
              "severity": "medium",
              "identifiers": {
                "issue": "2432",
                "summary": "3rd party CORS request may execute",
                "CVE": [
                  "CVE-2015-9251"
                ]
              }
            },
            {
              "info": [
```

Figure 26 Retire.js results

In the previous example we run retire.js and formatted the JSON result with jq. We can see in the results that retire.js indicates which file has a vulnerability, the description of the security issue and its severity.

3.2.2.4 Creating the Jenkins job

Now that we have the tools configured correctly, we can create the Jenkins job. In Jenkins Dashboard, we select the option to create a Multibranch Pipeline object.

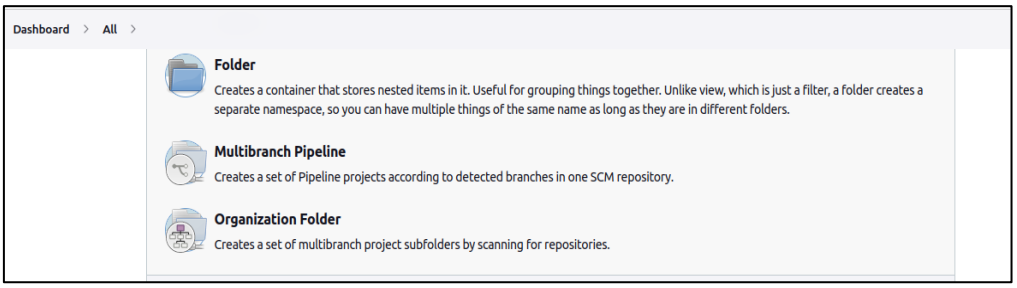

Figure 27 Multibranch pipeline item

Once created, we will configure it starting with the Branch sources. We will choose Bitbucket as the code repository, enter the credentials (OAuth consumer from the previous section), the owner of the repository and Jenkins will load all the repositories available.

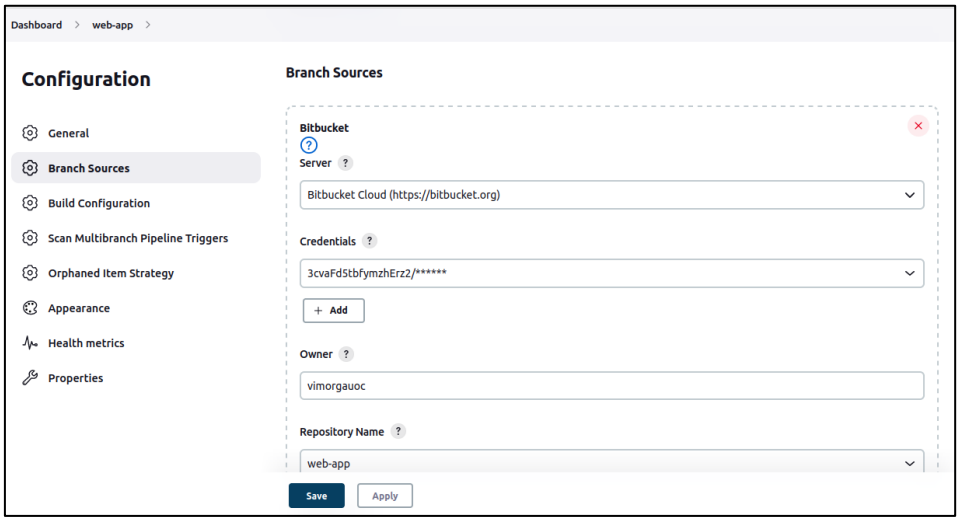

Figure 28 Bitbucket source configuration in Jenkins

For build configuration we specify that we want to use a Jenkinsfile that is going to be located in the root of the project. A Jenkinsfile is a groovy file that contains the steps to build a pipeline in Jenkins.

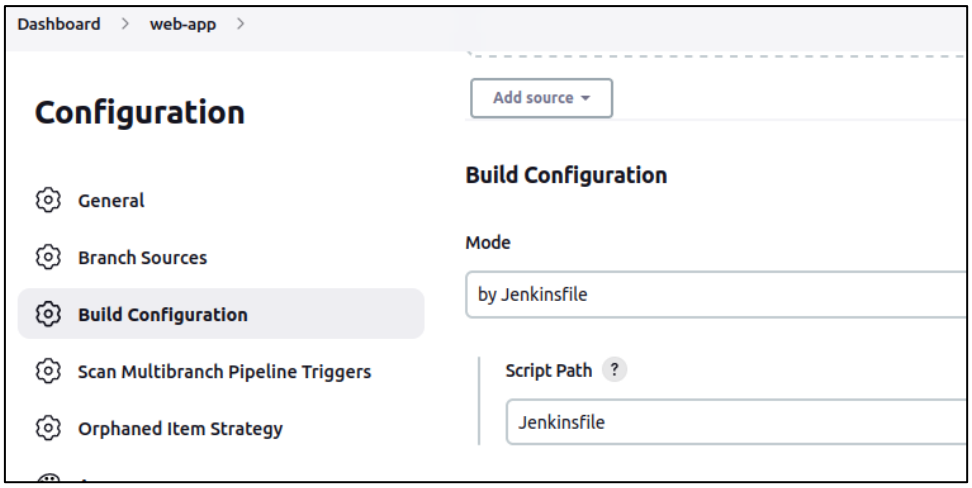

Figure 29 Jenkins build configuration

The Jenkinsfile is going to be divided into 4 stages. In the first stage (Checkout) we are going to inform Bitbucket cloud that a build is in process and retrieve the source code from the repository

```
stage('Checkout') {
    steps {
        bitbucketStatusNotify(buildState: 'INPROGRESS')
        checkout scm
    }
}
```

Figure 30 Checkout stage in Jenkinsfile

In the second stage (SAST) we are going to use the SonarQube plugin to build our sample project and scan it with our SonarQube server instance.

```
stage('SAST') {
    environment {
        scannerHome = tool 'SonarQube'
    }
    steps {
        script{
            sh "npm install"
            withSonarQubeEnv('sonarqube') {
                sh """
                ${scannerHome}/bin/sonar-scanner \
                -D sonar.projectKey=demo \
                -D sonar.sources=.\
                """
            }
        }
    }
}
```

Figure 31 SAST stage in Jenkinsfile

After the scan is finished, in the third stage we call Quality Gates to check if the scan complies with our security standards, if not, the build will fail.

```
stage('Quality gate') {
        steps {
            waitForQualityGate abortPipeline: true
        }
}
```

Figure 32 Quality gates stage in Jenkinsfile

In the fourth stage (SCA) we run retire.js and save the results in a JSON file. Since retire.js doesn't have a module like Quality Gates, we use jq to extract the data from the results and set a security standard for the job to be successful. In this case, if retire.js finds a vulnerability which severity is high or critical the job will fail.

```
stage('SCA') {
      steps {
         script {
            sh 'retire --outputformat json --exitwith 0 > SCAResults.json'
            sca_results = sh(
               script: '^cat SCAResults.json | jq \' .data[].results[].vulnerabilities[] |^
               select(.severity == "high" or .severity == "critical")  \' ',
               returnStdout: true
               )

            if (sca_results != null && sca_results.trim() != '') {
               error('SCA detected severe vulnerabilities')
            }
         }
      }
}
```

Figure 33 SCA stage in Jenkinsfile

Finally, the job saves the JSON file with the SCA results (SAST results are stored in SonarQube server) and notifies Bitbucket if the build has failed or not.

```
post {
   always {
         archiveArtifacts artifacts: 'SCAResults.json', onlyIfSuccessful: false
   }

   success {
         bitbucketStatusNotify(buildState: 'SUCCESSFUL')
   }

   aborted {
         bitbucketStatusNotify(buildState: 'FAILED')
   }

   failure {
         bitbucketStatusNotify(buildState: 'FAILED')^
   }
}
```

Figure 34 Post actions in Jenkinsfile

3.2.3. Implementation in the preproduction and production tiers

In this tier we are going to implement IAST and DAST technology. As we saw in the previous chapter, we are going to install Contrast IAST first, so when we run OWASP ZAP tests we can take advantage of those requests for IAST scan.

3.2.3.1 Contrast IAST set up

In this project we are going to use Contrast Community edition. First, we have to fill in a form on their website (45) to sign up:

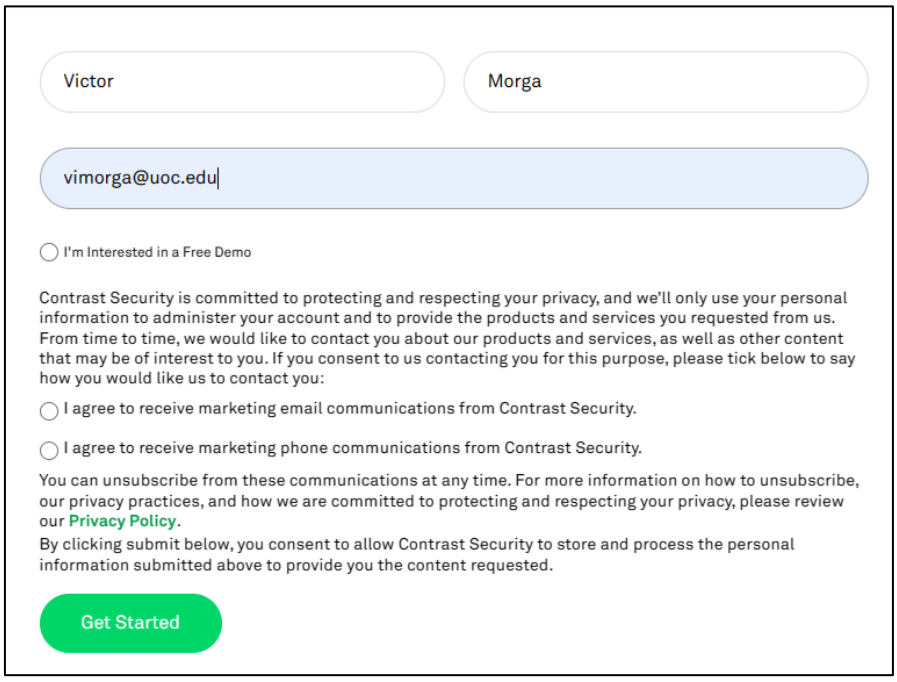

Figure 35 Form to register on Contrast Community Edition

After we sign up, we have to install a Contrast agent in our server to instrumentalize our application. Since IAST is language dependent, we have to choose the language accordingly, which in our case is Node.

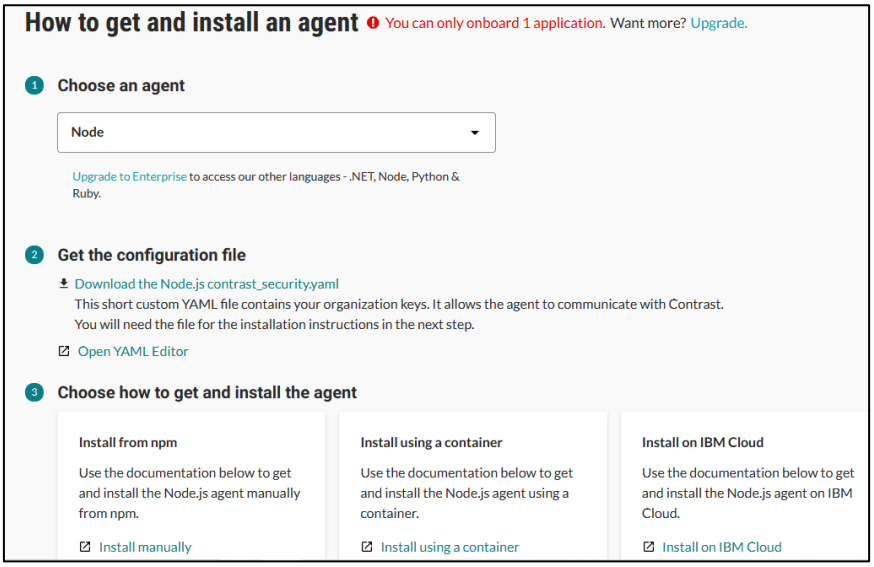

Figure 36 Instructions to install Contrast agent

When the language is selected, we have to create a configuration file in our project so the agent can communicate with the server. The file is in YAML format, and it contains the credentials for the API.

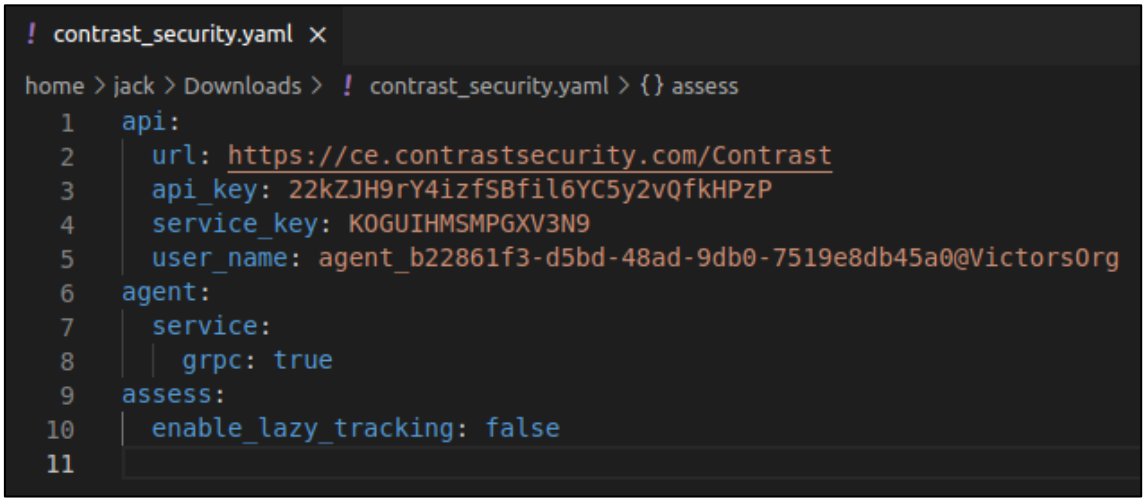

Figure 37 YAML configuration file for Contrast

Now it is time to install the agent. For Node, we use npm to install the contrast agent with the command `npm install @contrast/agent`.

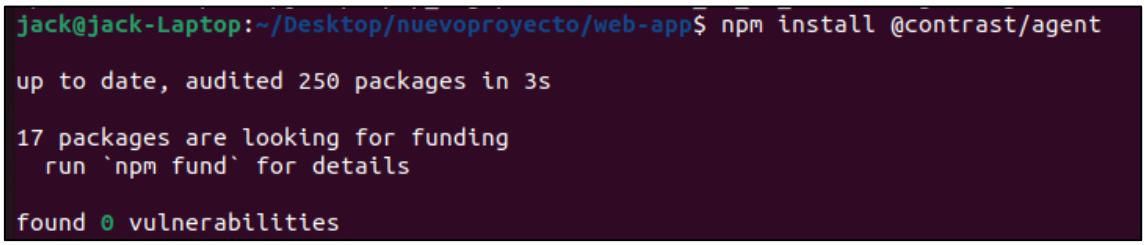

Figure 38 Contrast IAST agent installation

After the installation we can find the dependency in the project's package.json. In that file we create a script to run our app with Contrast's Agent using the command `node -r @contrast/agent server.js`.

Figure 39 package.json file with the script to run the app with Contrast's agent

With the script done, now we just have to launch our app with the previous script using `npm run contrast.`

```
jack@jack-Laptop:~/Desktop/nuevoproyecto/web-app$ npm run contrast

> contrast
> node -r @contrast/agent server.js


@contrast/agent 4.29.1
---------------------------------------
Example app listening at http://:::8081
```

Figure 40 Launching app with Contrast's script

3.2.3.2. OWASP set up

For OWASP ZAP we are going to use a docker image to run the tests. First, we create a pipeline in Jenkins with the following code.

In the first stage we pull the image from Docker, start the container and create a working directory.

```
stages {
    stage('OWASP ZAP docker image') {
        steps {
            script {

                sh 'docker pull owasp/zap2docker-stable'
                sh """
                docker run -dt --name owasp \
                owasp/zap2docker-stable \
                /bin/bash
                """
                sh """
                    docker exec owasp \
                    mkdir /zap/wrk
                """
            }
        }
    }
```

Figure 41 OWASP ZAP docker image is pulled and executed.

After that, we run a full scan against our target and create a report. Depending of the results, the command's exit code can break the pipeline, so we have to use a try/catch block if we want to continue despite the results.

```
stage('Scanning') {
    steps {
        script{
            try{
                sh """
                    docker exec owasp \
                    zap-full-scan.py \
                    -t ${params.TARGET} \
                    -x report.xml \
                    -I
                """
            }catch(e) {}
        }
    }
}
```

Figure 42 Command to run the OWASP ZAP scan.

Once the scan is finished, we get retrieve the results from the container and archive the XML file in Jenkins.

```
    stage('Report'){
        steps {
            script {
                sh '''
                    docker cp owasp:/zap/wrk/report.xml ${WORKSPACE}/report.xml
                '''
            }
        }
    }
}
post {
        always {
            archiveArtifacts artifacts: 'report.xml', onlyIfSuccessful: false
            sh '''
                docker stop owasp
                docker rm owasp
            '''
        }
    }
```

Figure 43 Retrieving and saving the results from OWASP ZAP's scan

3.3 Testing the pipelines

For testing the pipelines, a simple Node.js web app is going to be used. This app contains two vulnerabilities: a hardcoded password and a SQL injection (The code is in the annex).

First, we are going to test the SAST and SCA scans in the integration tier.

To test the process, we are going to create a new branch from the main branch in the repository, modify a file and open a pull request. Once the pull request is opened, we can see that a Jenkins build has been triggered.

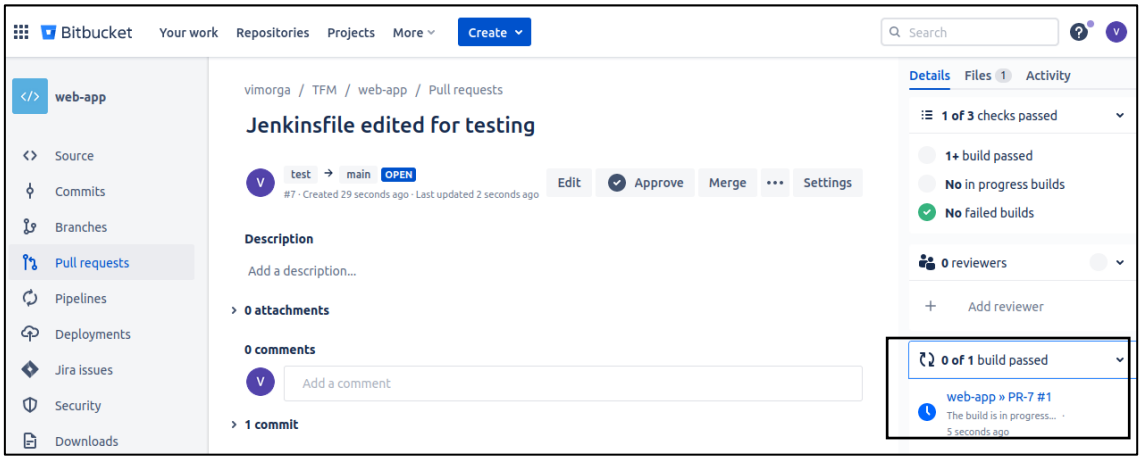

Figure 44 Build trigger in Bitbucket

If we inspect the Jenkins job, we find that it has failed because of Quality Gates

```
Checking status of SonarQube task 'AYVgpttB7GYTx9qaWzFo' on server 'sonarqube'
SonarQube task 'AYVgpttB7GYTx9qaWzFo' status is 'PENDING'
SonarQube task 'AYVgpttB7GYTx9qaWzFo' status is 'SUCCESS'
SonarQube task 'AYVgpttB7GYTx9qaWzFo' completed. Quality gate is 'ERROR'
```

Figure 45 Jenkins job failed due to Quality Gates

This is because SonarQube had find two major security issues in our code (but the SQL injection wasn't detected).

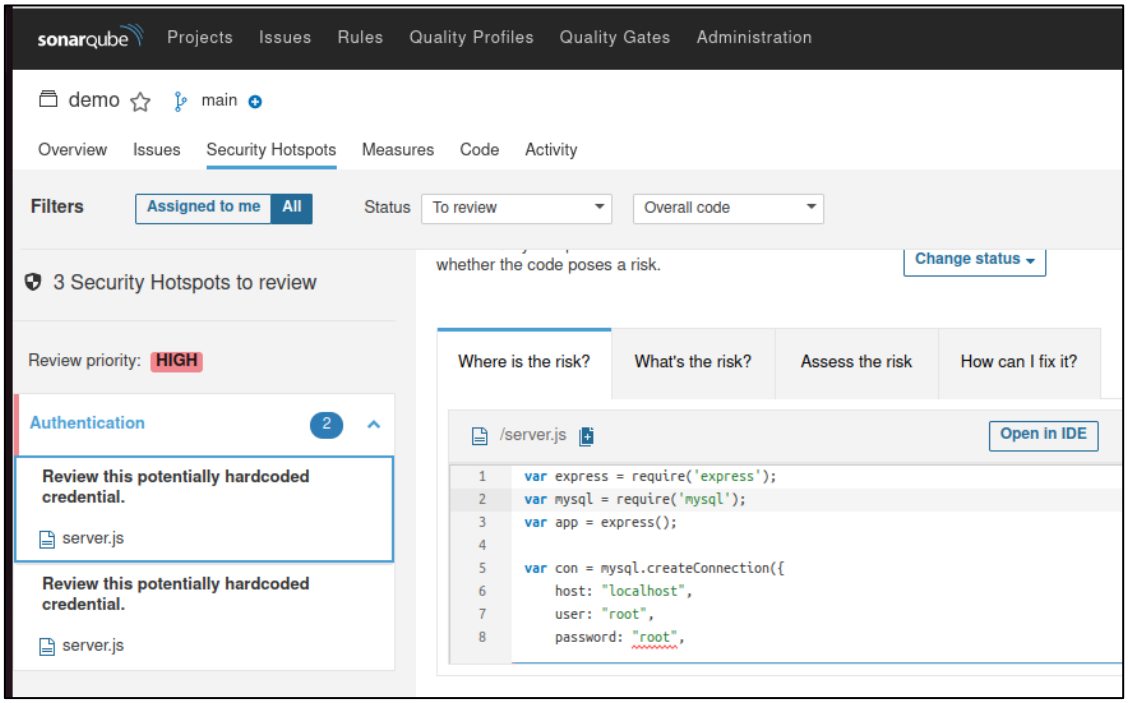

Figure 46 Security issues in SonarQube

If we fix the issues and commit the changes, the code in the pull request will be updated and trigger the Jenkins job once again. This time the build will be successful, and the conditions to be able to merge the branch will be met.

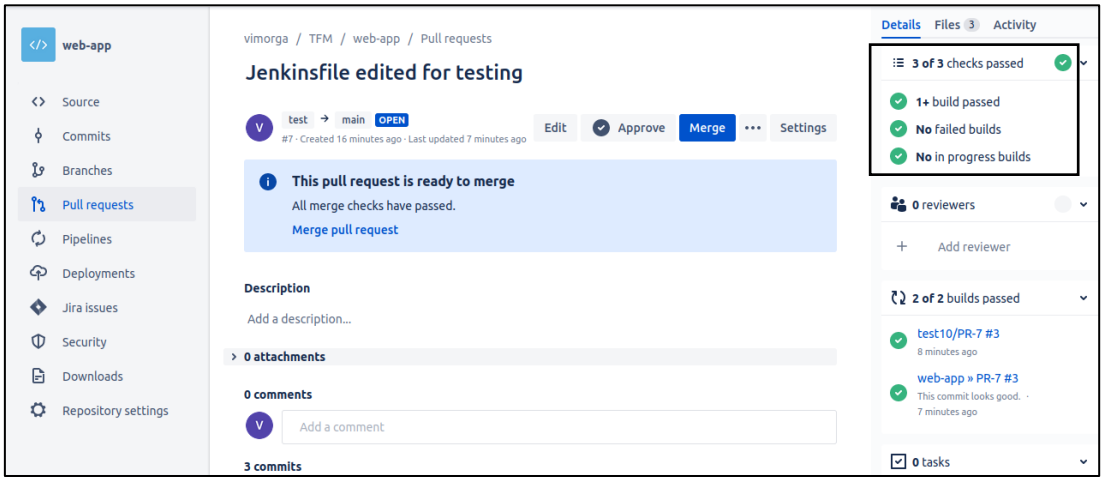

Figure 47 Conditions passed for merging a branch

If we inspect the Jenkins Job will see that it was successful and the SCA scan results were saved.

Figure 48 Successful build in Jenkins

Now we are going to test DAST and IAST in preproduction and production tiers. With the Node application running with contrast agent, we execute the Jenkins job with Contrast's agent.

After the job is finished, we get the following report

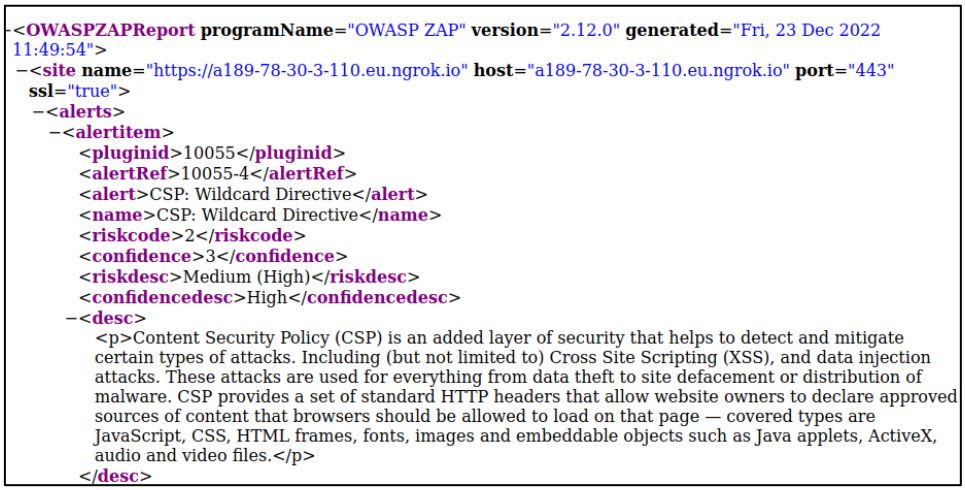

Figure 49 DAST results

The DAST results found vulnerabilities in our server configuration, but not in the code. Let’s check Contrast:

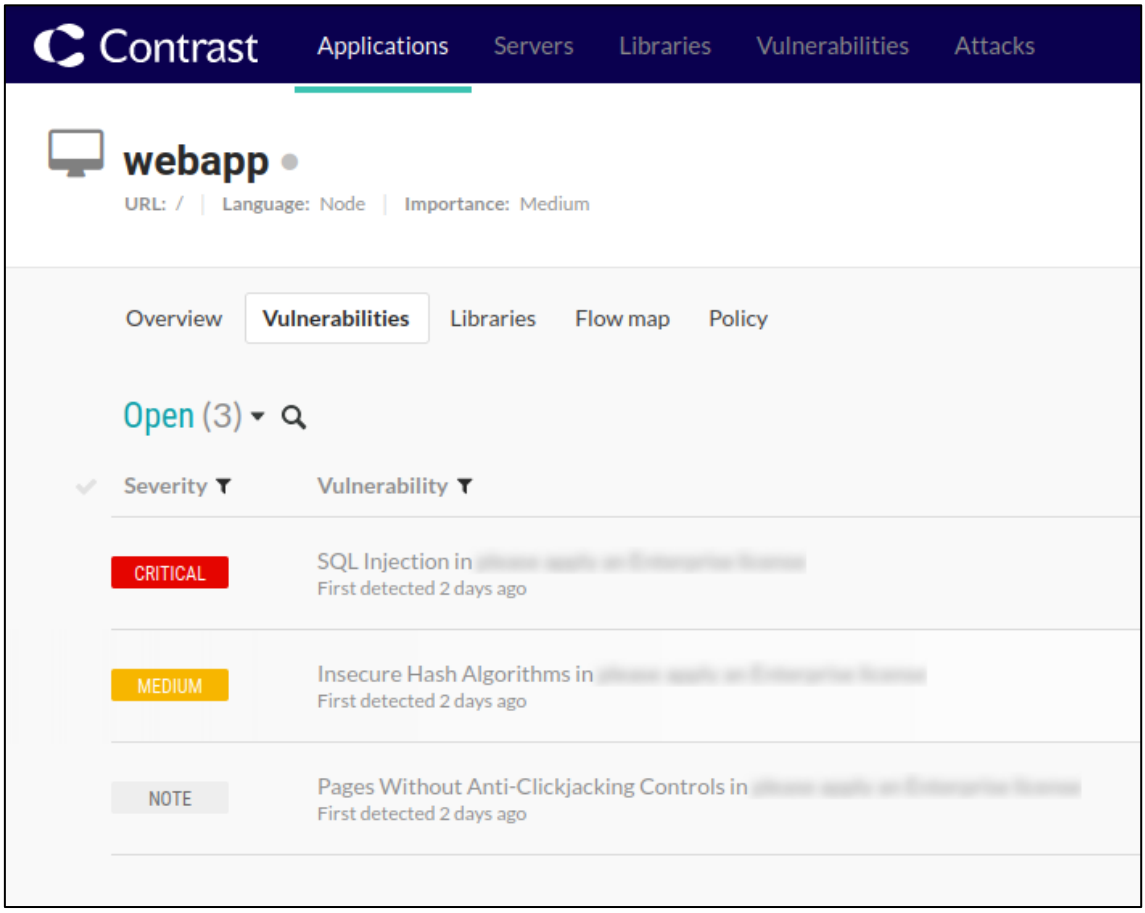

Figure 50 Contrast IAST results

We can see that Contrast IAST has detected the SQL injection vulnerability, but due that is a free plan it does not tell us where it is.

4. Conclusions

We can improve the security of our applications using these technologies. With the correct set up it can save a lot of time and money to software development companies.

However, the tools that are free or open source are not 100% reliable. The SQL injection was undetected until Contrast IAST detected the issue.

For an independent developer or small company with limited budget these free tools can improve their security detecting issues that they are not aware. But for companies that can afford it, it is better to use commercial scanners.

Commercials solutions can integrate SAST, DAST, IAST and SCA in the same solution, thus reporting the security issues in the same space instead of multiple files across the environment, which helps the security engineers to organize the information and save their time for other tasks.

5. Lessons learned

In this paper we learned about the automated code analysis tools and its categories.

We learned that SAST can scan the source code of our application and find vulnerabilities using regular expressions.

We learned that DAST uses a set of penetration tests to find vulnerabilities in a running application without knowing how our application works.

We learned that IAST uses instrumentalization to monitor the application in real time and inform the developers in case of suspicious or problematic requests.

We learned that SCA scans third party libraries for vulnerabilities and licensing compliance.

We learned how to set up and configure a modern CI/CD environment and integrate the previous technologies to secure our product and automate the process.

6. Future work

In this section we will explain how the current environment can be expanded to make the product more secure or automate work in order to reduce the workload of the developers, but due to the time limitation these features were out of scope.

Create unit tests for code coverage

We know that SonarQube has a module called Quality Gates that measures the quality of the code. If the code does not comply with our quality standards, the Jenkins job fails.

One of the measures of Quality Gates was code coverage by unit tests. Unit tests are used to make sure that the pieces of our code do what they are intended to do. With SonarQube we can measure the percentage of code that is tested, this way we can be sure that the code is reliable.

Update outdated third party libraries automatically

With SCA technology we are aware of which library versions are vulnerable. Commercial solutions can open pull request in our code repository to update those libraries to newer versions where the vulnerability has been fixed.

If our budget does not allow us to use commercial solutions, we could create our own bot that using the result from retire.js will search for the newer version of the vulnerable libraries.

Automating DAST scans

In the same way that we have automated SAST and SCA scans for each pull request that is opened for our main branch, the same can be done when a branch for a release is created.

When we create a new branch for the latest version of our product, we can trigger a Jenkins job to run a DAST scan for the build with the newer code, thus leveraging the workload of the developers

References

1. Forbes. “Alarming Cyber Statistics For Mid-Year 2022 That You Need To Know.” . [Website] 9/2022. https://www.forbes.com/sites/chuckbrooks/2022/06/03/alarming-cyber-statistics-for-mid-year-2022-that-you-need-to-know/?sh=10c668347864.

2. Github. [Website] 9/ 2022. https://github.com/.

3. Gitlab. [Website] 9/2022. https://about.gitlab.com/.

4. Bitbucket. [Website] 9/2022. https://bitbucket.org/.

5. Atlassian. "Jira". [Website] https://www.atlassian.com/software/jira.

6. OWASP. “Source Code Analysis Tools”. [Website] 10/2022. https://owasp.org/www-community/Source_Code_Analysis_Tools.

7. Gartner."Static Application Security Testing (SAST)". [Website] 10/2022. https://www.gartner.com/en/information-technology/glossary/static-application-security-testing-sast.

8. Mend. "SAST – All About Static Application Security Testing". [Website] 10/2022. https://www.mend.io/resources/blog/sast-static-application-security-testing/.

9. *Snyk. "Static Application Security Testing (SAST)".* [Website] 10/2022. https://snyk.io/learn/application-security/static-application-security-testing/.

10. *HCLSoftware."HCL AppScan CodeSweep".* [Website] 10/2022. https://www.hcltechsw.com/appscan/codesweep.

11. Checkmarx. "Checkmarx SAST". [Website] 10/2022. https://checkmarx.com/product/cxsast-source-code-scanning/.

12. Sonar. "SonarQube". [Website] 10/2022. https://www.sonarsource.com/products/sonarqube/.

13. CyberRes. "Fortify Static Code Analyzer". [Website] 10/2022. https://www.microfocus.com/es-es/cyberres/application-security/static-code-analyzer.

14. *Microfocus."What is Dynamic Application Security Testing (DAST)?".* [Website] 10/2022. https://www.microfocus.com/en-us/what-is/dast.

15. *Astra."15 Best Dynamic Application Security Testing(DAST) Software in 2022".* [Website] 10/2022. https://www.getastra.com/blog/security-audit/top-dast-tools/.

16. *OWASP. "OWASP ZAP".* [Website] 10/2022. https://owasp.org/www-project-zap/.
17. Acunetix. [Website] 10/2022. https://www.acunetix.com/.
18. Rapid7. "InsightAppSec". [Website] 10/2022. https://www.rapid7.com/products/insightappsec/.
19. Noname Security "What is Dynamic Application Security Testing (DAST)?". [Website] https://nonamesecurity.com/learn-what-is-dast.
20. Hdivsecurity "What is IAST? All About Interactive Application Security Testing". [Website] 10/2022. https://hdivsecurity.com/bornsecure/what-is-iast-interactive-application-security-testing/.
21. Acunetix. "Acusensor". [Website] 11/2022. https://www.acunetix.com/vulnerability-scanner/acusensor-technology/.
22. Contrast. [Website] 10/2022. https://www.contrastsecurity.com/glossary/interactive-application-security-testing.
23. Checkmarx "CxIAST". [Website] 10/2022. https://checkmarx.com/product/cxiast-interactive-code-scanning/.
24. Synopsys. "Software Composition Analysis". [Website] 10/2022. https://www.synopsys.com/glossary/what-is-software-composition-analysis.html.
25. Checkmarx. "Checkmarx SCA". [Website] 10/2022. https://checkmarx.com/cxsca-open-source-scanning/.
26. Github "retire.js". [Website] https://retirejs.github.io/retire.js/.
27. Mend. "Mend SCA". [Website] 10/2022. https://www.mend.io/sca/.
28. Jenkins. "Checkmarx plugin". [Website] 11/2022. https://www.jenkins.io/doc/pipeline/steps/checkmarx/ .
29. Visual Studio Code. [Website] 11/2022. https://code.visualstudio.com/.
30. Jenkins. [Website] 10/2022. https://www.jenkins.io/.
31. Ubuntu. [Website] 11/2022. https://ubuntu.com/.
32. Oracle. "Download Java". [Website] 11/2022. https://www.oracle.com/java/technologies/downloads/.
33. Node.js. [Website] 11/2022. https://nodejs.org/.
34. Docker. [Website] https://www.docker.com/.
35. Ngrok. [Website] 11/2022. https://ngrok.com/.
36. npm. [Website] https://www.npmjs.com/.

37. Github "jq". [Website] 11/2022. https://stedolan.github.io/jq/.
38. JetBrain "Intellij IDEA". [Website] 11/2022. https://www.jetbrains.com/idea/.
39. Github "bitbucket branch source plugin". [Website] 11/2022. https://github.com/jenkinsci/bitbucket-branch-source-plugin.
40. github "bitbucket build status notifier plugin". [Website] 11/2022. https://github.com/jenkinsci/bitbucket-build-status-notifier-plugin.
41. AWS "Amazon EC2". [Website] 11/2022. https://aws.amazon.com/ec2/?nc1=h_ls.
42. Azure "Virtual Machines·. [Website] 11/2022. https://azure.microsoft.com/en-us/products/virtual-machines/.
43. Github. "SonarQube Scanner for Jenkins". [Website] 11/2022. https://github.com/jenkinsci/sonarqube-plugin.
44. Sonarqube. "Quality Gates". [Website] 11/2022. https://docs.sonarqube.org/latest/user-guide/quality-gates/.
45. Contrast Community Edition. [Website] 11/2022. https://www.contrastsecurity.com/contrast-community-edition.
46. *Cirt.net "Nikto".* [Website] 10/2022. https://cirt.net/Nikto2.
47. Apache Maven Project. [Website] 11/2022. https://maven.apache.org/.

Annex

- Jenkinsfile for SAST/SCA integration:

```
pipeline {
  agent any
  stages {
    stage('Checkout') {
      steps {
        bitbucketStatusNotify(buildState: 'INPROGRESS')
        checkout scm
      }
    }

    stage('SAST') {
      steps {
        withSonarQubeEnv(installationName: 'sonarqube', credentialsId: 'SonarQubeToken')
{
          sh 'mvn clean verify sonar:sonar -Dsonar.projectKey=SonarQube'
        }
      }
    }

    stage("Quality gate") {
      steps {
        waitForQualityGate abortPipeline: true
      }
    }

    stage('SCA') {
      steps {
        script {
          sh 'retire --outputformat json --exitwith 0 > SCAResults.json'
          sca_results = sh(
            script: ' cat SCAResults.json | jq \' .data[].results[].vulnerabilities[] |
select(.severity == "high" or .severity == "critical")  \' ',
            returnStdout: true
          )

          if (sca_results != null && sca_results.trim() != '') {
            error('SCA detected severe vulnerabilities')
          }
        }
      }
    }
  }
```

```
  post {
    always {
      archiveArtifacts artifacts: 'SCAResults.json', onlyIfSuccessful: false
    }

    success {
      bitbucketStatusNotify(buildState: 'SUCCESSFUL')
    }

    aborted {
      bitbucketStatusNotify(buildState: 'FAILED')
    }

    failure {
      bitbucketStatusNotify(buildState: 'FAILED')
    }
  }
}
```

- Sample Node.js code with vulnerabilities for tests:

```
var express = require('express');
var mysql = require('mysql');
var app = express();

var con = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "root",
    database: "restapi"
});

var password = "mypassword"

app.get('/', function(req, res) {
    res.send('Hello World');
})
app.get('/hi', function(req, res) {
    const user_id = req.query.id;
    var sql = "SELECT * FROM users WHERE id=" + user_id;
    con.query(sql, function(err, result) {
        if (err) throw err;
        res.send(result);
    });
})

var server = app.listen(8081, function() {
    var host = server.address().address
    var port = server.address().port

    console.log("Example app listening at http://%s:%s", host, port)
})
```