NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ANDRÉ JORGE MARTINS SOUSA

Bsc in Computer Science

CYBERSECURITY IN THE APPLICATIONAL CONTEXT

KEYCLOAK, KONG AND CODE VERIFIER

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

December, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CYBERSECURITY IN THE APPLICATIONAL CONTEXT

KEYCLOAK, KONG AND CODE VERIFIER

ANDRÉ JORGE MARTINS SOUSA

Bsc in Computer Science

Adviser: Tiago Filipe Vieira Epifânio
Software Developer, Link Consulting

Co-adviser: António Maria Lobo César Alarcão Ravara
Associate Professor, NOVA University Lisbon

Examination Committee

Chairs: João Ricardo Viegas da Costa Seco
Associate Professor, FCT-NOVA

João Manuel dos Santos Lourenço
Associate Professor, FCT-NOVA

Tiago Filipe Vieira Epifânio
Software Developer, Link Consulting

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2023

Cybersecurity in the applicational context
KEYCLOAK, KONG AND CODE VERIFIER

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.5) [65].

To the beginning of a new life.

Acknowledgements

I would like to start by thanking my supervisor, Professor António Ravara, for all his help and support, without which this thesis would not have been possible.

Next, I would like to thank Luis Caetano, Pedro Samorrinha and Link Consulting for welcoming me with open arms and allowing me to carry out this process, always in good company. Always friendly and ready to help, they gave me all the help I needed, without which this thesis would not have even begun.

A special thanks to Tiago Epifânio, who closely accompanied me in this thesis, always ready to help, always available, even when he had a short time available. Without your tips and teachings everything would have been much more complicated, thank you very much for your support.

To all my friends, who started university with me and stayed with me until the end, a heartfelt thank you for all the moments of study and work, but also of fun, without which none of this would have been possible.

To all my family, but particularly my parents, brother and grandmother, a big thank you for all your support and understanding, for accompanying me through this process and for making it all possible.

To my girlfriend, thank you for all your support and for always being there when I needed you, never letting me down or discouraging me, and always believing in me. Without you, it would have been much more complicated and gray, because with you, life gains colour.

”

“The computer was born to solve problems that did not exist before.”

— Bill Gates

Abstract

Cybercrime has increased over the past few years, as has the number of victims and the amount of data lost and stolen. This escalation in cybercrime has led to greater concern about cybersecurity, which has resulted in the creation of norms, rules and standards to guide and protect users in cyberspace. This special concern for cybersecurity has also reached Link Consulting, which, as an IT company, endeavours to guarantee the security of its products.

Link Consulting, responsible for suggesting the topic of this dissertation, is a technology company that seeks to offer its customers IT solutions that not only fulfil their requirements, but also guarantee reliability and trust. The Egov department is one of the various departments in the company that will be the subject of this dissertation. The most significant difference between Egov and the other departments is that it focuses on work in the public sector only.

The department currently has a reference architecture to guide programmers in their work. With this increase in concern for Link, there was an interest in reinforcing and guaranteeing the security of the architecture. It is possible to identify some tools and possible vulnerabilities. These possible points of failure were studied, identifying the causes of the problem and possible solutions. To help Link mitigate the existing vulnerabilities in its architecture, a mechanism was created to automate the verification of these points of failure, complementing the current mechanisms with another tool that could be beneficial to the company.

After analysing the entire architecture and mitigating the possible existing vulnerabilities, it was then possible to obtain a plugin that can not only identify the vulnerabilities present in Keycloak, but also check for them and report them to users. The result allows users to visualize in Jenkins (or in an output file) the result of the checks made on Keycloak, showing the vulnerabilities found and the proposed solution to them. Since Keycloak is widely known and used, in order to allow users to have greater confidence in it, and considering the results obtained to be of public use, it is intended to publish the results as open-source, so that they can be implemented by other users and thus minimize the chance of vulnerabilities in their Keycloak instances as well. Therefore, the vulnerabilities present

in the reference architecture are considered to be mitigated, and it has been possible to take another step towards guaranteeing its security.

Keywords: cybercrime, cybersecurity, vulnerabilities, architecture, plugin

Resumo

O cibercrime aumentou ao longo dos anos, aumentando também a quantidade de lesados e de dados perdidos e roubados. Este escalar do cibercrime resultou numa maior preocupação com cibersegurança, que resultou na criação de normas, regras e padrões com o intuito de guiar e proteger os utilizadores no ciberespaço. Esta atenção especial para com a cibersegurança também chegou à Link Consulting que, como empresa informática, procura garantir a segurança dos seus produtos.

A Link Consulting, responsável pela sugestão do tema desta dissertação, é uma empresa do ramo tecnológico que procura oferecer aos seus clientes soluções informáticas que, não só preencham os requisitos dos clientes, como também garantam fiabilidade e confiança. De entre os vários departamentos existentes na empresa, é com o departamento de Egov que será desenvolvido o tema. A diferença mais significativa entre Egov e os restantes departamentos, é que este se foca em trabalhos apenas no setor público.

Atualmente, este departamento conta com uma arquitetura de referência que permite guiar os programadores na sua tarefa. Com este aumento da preocupação da Link, surgiu o interesse em reforçar e garantir a segurança da arquitetura. Na mesma é possível identificar algumas ferramentas e possíveis vulnerabilidades. Esses possíveis pontos de falha foram estudados, identificando as causas do problema e possíveis soluções. Para ajudar a Link a mitigar as vulnerabilidades existentes na sua arquitetura, foi criado um mecanismo de automatização de verificação desses pontos de falha, complementando os mecanismos atuais com outra ferramenta que possa ser benéfica para a empresa.

Após analisada toda a arquitetura, e mitigadas as possíveis vulnerabilidades existentes, foi possível, então, obter um plugin que consegue, não só, identificar as vulnerabilidades presentes no Keycloak, como também verificar as mesmas e informar os utilizadores. O resultado permite que os utilizadores visualizem no Jenkins (ou num ficheiro de output) o resultado das verificações feitas ao Keycloak, mostrando as vulnerabilidades encontradas e a solução proposta para as mesmas. Uma vez que o Keycloak é largamente conhecido e utilizado, para permitir uma maior confiança por parte dos utilizadores, no mesmo, e considerando os resultados obtidos como sendo de utilidade pública, tenciona-se publicar os resultados como open-source, para poderem ser implementados por outros

utilizadores e dessa forma minimizarem a probabilidade de existência de vulnerabilidades nas suas instâncias de Keycloak também. Assim sendo, consideram-se mitigadas as vulnerabilidades presentes na arquitetura de referência, e foi possível dar mais um passo no caminho de garantir a segurança da mesma.

Palavras-chave: cibercrime, cibersegurança, vulnerabilidades, arquitetura, plugin

Contents

List of Figures

1 Introduction

1.1 Context

With the advancement of technology, its globalization and the increased ease of purchasing technological products, the number of technology users has seen a growth. As a result, the number of Internet users has also experienced an escalation, resulting in a new possibility of crime: the cybercrime. This type of crime has become more common over the years, with worldwide cyberattacks set to increase by 125% by 2021. That same year saw nearly 1 billion emails exposed, affecting 1 in 5 Internet users. Since 2001, the victim count has jumped from 6 victims per hour to 97, a 1517% rise over 20 years [55].

These attacks resulted in millions of euros lost, private information made public, among other constraints, so the focus on cybersecurity has also increased over the years. Then, in 2014, the National Cybersecurity Center emerged with the mission of contributing to a free, reliable and secure use of cyberspace of national interest [38]. This aims to protect citizens who use the Internet by providing various indications, awareness, and training, but also to protect companies that work in the national cyberspace, also through these same mechanisms and yet another, the National Reference Framework in Cybersecurity. The latter seeks to help organizations reduce the risk associated with cyber threats, providing the basis for any entity to meet the minimum security requirements of networks and information systems, in its various components. Namely, in the identification, protection, detection, response and recovery to cyber incidents, including the necessary organization for their management [47].

1.2 Motivation

Cybersecurity is a major concern to companies, which seek to protect their clients and projects in the most secure way possible. Companies continuously seek to improve the security of their products, knowing that in case of loss the cost will be much higher. This is a concern that affects Link as well, its objective is to guarantee the security of the products it creates. Link currently has a reference architecture, in which requests pass

through a load-balancer and are forwarded to the intended services. Among these are the services that support the API's exposed by Keycloak and Kong. Keycloak is responsible for authenticating users, so that users without access can not gain access to services. Once authenticated, requests are directed to Kong, which is an API gateway, responsible for directing incoming requests to the respective services.

1.3 Problem

If users manage to gain access to Keycloak, they will be able to gain access to the services and maybe escalate the privileges granted to them. Kong can also have vulnerabilities that escape programmers, such as the possibility of accessing Kong and gaining access to services protected by it. In addition to these, there are other possibilities of vulnerabilities related to the code itself created by programmers, such as the presence of explicit credentials in programming, or other errors that may allow the creation of vulnerabilities in applications.

1.4 Objectives

The objective of this work will be to mitigate the previously mentioned vulnerabilities and will go through the deconstruction of the architecture into smaller parts: Keycloak, Kong and the code itself. The aim is to obtain a solution that automates the process of testing these tools and the code, using pipeline scans for both tools, and code analysis tools for the code. The expected solution will mainly be a set of scans that will automatically check, upon code changes, the Keycloak settings to ensure that the current configuration is no less secure than the previous one, and potentially even more secure. These scans will be complemented by the use of static code verification tools. The expected solution will be able to guarantee, regardless of the version of the tool in use, the security of the tools against current known vulnerabilities. Since these scans will not carry out penetration tests, it will not be possible to discover new vulnerabilities, but rather ensure the mitigation of known vulnerabilities at the time of writing the document.

1.5 Contributions

The solution, namely the pipeline scan, will be an added value for users and companies, who will be able to access these scans free of charge and also guarantee the security of their Keycloak instance. To date there are no similar tools, so it is considered that the solution created will be very useful, filling a gap in the market, and helping developers to obtain the best and safest configuration possible of Keycloak. By ensuring a secure configuration of this tool and guaranteeing a code that complies with good programming practices, tested by tools mentioned in following chapters, it will be possible to present greater certainty and confidence in the security of the products created by the company.

1.6 Document Structure

This document is organised into 6 main chapters, which will guide the course of this dissertation so that there are no doubts at the end. The first chapter, Technical Background, will be responsible for presenting some concepts that are considered important for understanding the dissertation. This will cover various methodologies for testing applications, in order to gain a better understanding of the existing approaches and realise whether the choice made was really the right one. Some existing tools will also be presented, both for analysing code and for helping in the software development process, which will be referenced later in the document. Finally, some existing tools in the reference architecture will be analysed, as well as the plugin used in Jenkins, in order to understand the solution adopted.

The next chapter will look at the vulnerabilities that exist in Keycloak, in the code, as well as various testing tools that could be useful in the expected solution, or even be feasible alternatives. Finally, two tools will be presented that make it possible to analyse the presence of vulnerabilities by, among other methods, searching for existing Common Vulnerabilities and Exposures (CVE). The third chapter will look at the plugin that is expected to be produced, presenting some concepts and objectives, followed by an analysis of the architecture to see if there are any other vulnerabilities that can be mitigated. The next chapter will then go into detail about the work carried out, explaining what was done and how it was done, as well as why other paths were not followed. The problems encountered, which prevented certain paths, will also be mentioned and explained.

This will be followed by an experimental evaluation to ensure that the expected results have been obtained and that the solution behaves as expected. This will be followed by a conclusion to consolidate all the information and ensure that any doubts have been clarified and that the proposed objective has been fulfilled. At the end of reading the document, hopefully, there will be no doubts about the fulfilment of the objectives, the usefulness of the solution, or any other question.

2

Technical Background

A company is an entity that largely depends on its reputation in the market, which is why it is very important to present quality in the products it sells to customers. Some points that affect the quality of the generated product are, for example, product safety, reliability, security, and longevity, so it is important to ensure that the final product has as few flaws, errors, vulnerabilities as possible. For that, there are several ways to analyse the generated applications and products, which differ in the way and when they act, and can be used in earlier stages of the development process, in intermediate and final stages.

A software is something complex, which requires the correct coordination of several components, in order to achieve a product that presents certain standards of quality and functionality. To ensure these requirements, some precautions are taken, such as verifications, monitoring and tests. Tests such as performance tests (load, stress, endurance, and spike testing), security, usability, or even compatibility tests as well [42].

These tests will make it possible to see if the product is ready to be presented to the customer and used around the world. If the software passes the tests, correct functioning is guaranteed, not only in normal use, but also in the case of mass use, which brings more load to the network and to the system, since it passed load, stress, etc. Testing is complex and extensive, however, in this document we are only concerned with the security of the application. We will assume that the application is working properly and correctly. This does not guarantee the security of the application, only its correct functioning, and it may still be vulnerable to intruders. Therefore, to guarantee the security of the application, we will have to carry out tests different from the ones mentioned above, since we intend to guarantee a different characteristic from the previous ones. We start from a point where tests have already been carried out involving a large amount of users and data, to guarantee functionality in the face of stress, cohesion tests were also carried out to ensure the cohesive functioning of the various modules in the application, and all the other tests necessary to guarantee all the other characteristics were also successfully completed.

That said, we can now focus on web application security. To this end, we will present below some ways to test the security of a web application. In the end, we will see which ways were chosen for use in the next phase of the dissertation, as well as the reason for

choosing these over others.

2.1 Static Application Security Testing

Static application security testing (SAST), is a methodology that analyses the internal structure, design, and code of a product to verify input-output flow and improve design, usability, and security [56]. This analysis is done with access to the code, so that the person responsible for the tests can create a set of situations in which possible vulnerabilities found in the source code are tested. Since the tests are done with access to the code, the idea of this method is to generate tests that verify possible weaknesses. The source code is analysed, the parts of the code that could lead to weaknesses are analysed in more detail, to create tests that verify whether that part of the code can generate an error, or an unexpected response, in the application. For example, in the case of the existence of a field in which numbers are expected, if characters are introduced, what will be the application's response? Sometimes, the simplest cases, which escape the programmer, are the ones that cause the most problems.

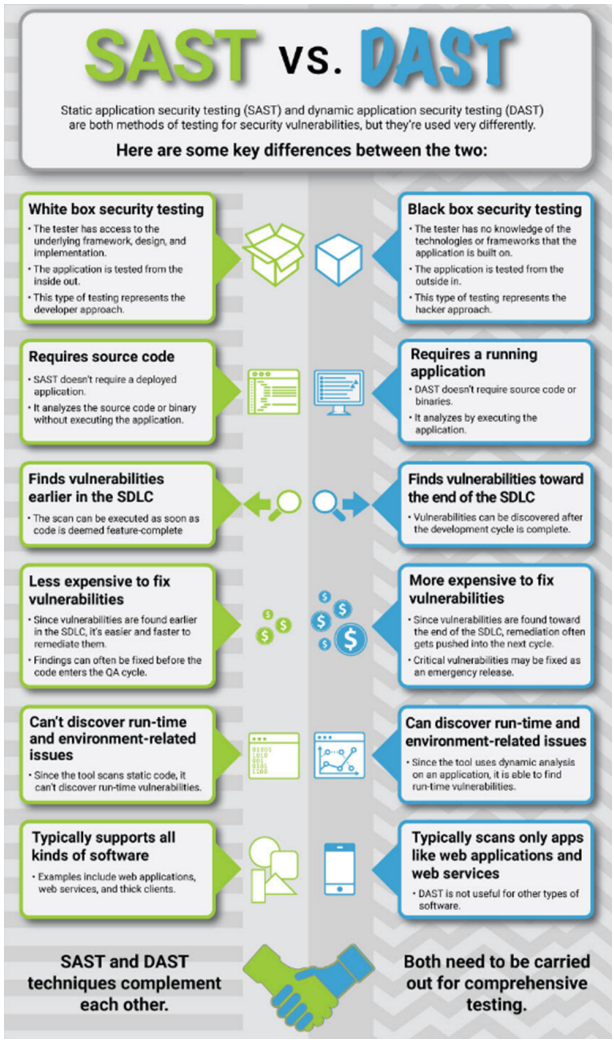
There are currently a lot of application testing applications, which check for situations such as buffer overflows, SQL injection, and other possible vulnerabilities in the application, that may lead to bigger problems or even application crash. This test can be performed, for example, by introducing SQL expressions in the field where another type of data would be expected. If the necessary precautions have not been taken with the SQL expression that accesses the data, present in the code, it may be possible to obtain more data from the database than would be expected, for this it is enough to introduce an SQL expression, in the field intended for other data, in the application, so that the expression present in the code is complemented with the expression introduced in the application and results in obtaining the data from the database. This is just an example of SQL injection testing; however, it is possible to identify the testing methodology: analyse the code, identify limit situations that may cause problems, create tests for these limit situations. Basically, the idea is to take the focus off the large data set and focus only on those limit situations that may have escaped the programmer, this way testing is done on the limits, instead of being performed with the data that is known to run well.

Since the tests are done to verify possible weaknesses in the code, it is possible to detect vulnerabilities at an earlier stage of the development cycle, even without the application fully functioning. This helps developers quickly resolve issues without breaking builds or passing on vulnerabilities to the final release of the application. SAST tools give developers real-time feedback as they code, and some tools point out the exact location of vulnerabilities and highlight the risky code [1]. Since vulnerabilities are discovered at an early stage of the development process, they do not require a deployed application, they do not require as much time and changes to resolve (compared to a later stage of the development cycle), it has a lower cost when compared to other testing forms. However,

since it does not need a deployed application, it cannot detect run-time and environment-related issues. It is also a way of detecting weaknesses that can easily issue false positives and, since the report is static, it becomes outdated quickly [1].

2.2 Dynamic Application Security Testing

Dynamic Application Security Testing (DAST) tests products during operation and provide feedback on compliance and general security issues. This methodology does not have knowledge of the application's internal interactions or designs at the system level, and does not even have access to the source code of the application (black box testing) and resorts to attacks and tests that simulate situations that users could execute (introduction of letters where numbers are expected, for example), but also simulate situations common among cyber-attacks [2]. After a DAST scanner performs these tests, it looks for results that are not part of the expected result set and identifies security vulnerabilities. Since this methodology does not refer to the Code, it produces fewer specific results, which require some more security knowledge to understand. A DAST scanner searches for vulnerabilities in a running application and then sends automated alerts if it finds flaws that allow for attacks like SQL injections, Cross-Site Scripting (XSS), and more. Since DAST tools don't have internal information about the application or the source code, they attack just as an external threat actor would [3]. DAST tools are equipped to function in a dynamic environment, so they can detect runtime flaws which SAST tools cannot identify.



[43]

Figure 2.1: Comparison between SAST and DAST.

As Figure 2.1 shows, SAST and DAST are quite different. According to the SAST methodology, the testers have knowledge about the bases of the program, allowing them to perform the testing from the code itself, not needing a running instance or the entire code, being possible to perform tests as the code is being developed. This last feature allows testing to be executed earlier, bringing lower costs, since errors are detected earlier. Despite these advantages, this methodology does not allow the detection of run-time vulnerabilities, since it does not require a running instance. On the other hand, DAST requires impartiality on the part of the tester, once no knowledge about the back-end of the application is required. This methodology needs an active instance, since it performs checks as if it were an attacker, thus allowing the discovery of a different type of vulnerabilities from those obtained with SAST. Since a functional instance is required, this methodology is pushed to the end of the development cycle, making the correction of possible vulnerabilities more costly. Both methodologies are valid and useful, however, when used together they allow for a more reliable and secure product [2].

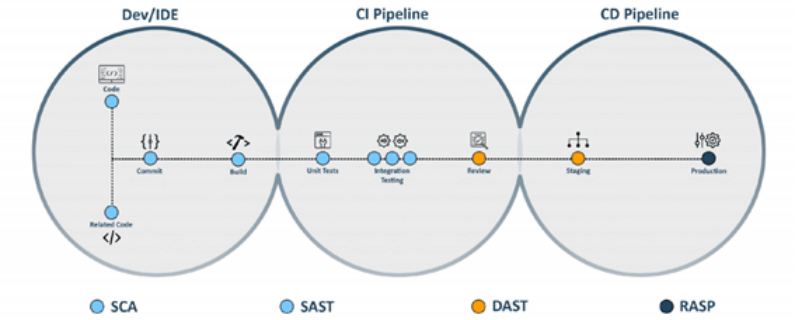
2.3 Interactive Application Security Testing

Interactive application security testing (IAST) works through software instrumentation, or the use of instruments to monitor an application as it runs and gather information about what it does and how it performs. This technique deploys agents and sensors in running applications and continuously analyze all application interactions initiated by manual tests, automated tests, or a combination of both to identify vulnerabilities in real time. In some tools, Software Composition Analysis (SCA) software is integrated, so known vulnerabilities in open-source components and frameworks are addressed [4]. Software composition analysis (SCA) evaluates security, license compliance, and code quality. This is achieved by comparing the version with a vulnerabilities list. This list contains known and common vulnerabilities, sorted by software version [5].

IAST effectively shifts testing left in the process timeline, so problems are caught earlier in the development cycle, reducing remediation costs and delays. IAST reports vulnerabilities in real-time, which means it does not add any extra time to the CI/CD pipeline, although many tools can be integrated into continuous integration (CI) and continuous development (CD) tools. Speed of results is also an advantage of IAST, which is best used in conjunction with other testing technologies [4].

2.4 Runtime Application Self Protection

Runtime Application Self Protection (RASP) is a security solution designed to provide personalized protection to applications by running on a server and kicking in when an application runs. It takes advantage of insight into an application's internal data and state to enable it to identify threats at runtime that may have otherwise been overlooked by other security solutions [6]. Developers can implement RASP by accessing the technology through function calls included in an app's source code, or by taking a completed app and put it in a wrapper that allows the app to be secured [60]. By deploying RASP, developers can identify vulnerabilities within their applications, since this methodology monitors the inputs, outputs, and internal state of the application it is protecting. It intercepts all calls from the app to a system, making sure they're secure, and validates data requests directly inside the app. Additionally, the RASP solution can block attempts to exploit existing vulnerabilities in deployed applications. When a security event in an app occurs, RASP takes control of the app and addresses the problem. RASP differs from other cybersecurity solutions in its level of focus on a single application. This focus enables it to provide several security benefits like visibility into Application-Layer Attacks, Zero-Day Protection, Lower False Positives, Easy Maintenance, Flexible Deployment, and Cloud Support [6].



[7]

Figure 2.2: Comparison between SAST, DAST and RASP.

As can be observed in Figure 2.2, SCA (Software Composition Analysis) techniques, as well as SAST, are useful at the beginning of the development process, being present from an early age. These are complemented, at a later stage of the process, by DAST techniques. When the product is finalized and ready to move into a production environment then RASP techniques can be applied to monitor and secure the application.

2.5 Sonarqube

SonarQube [8] is an open-source tool that empowers developers and organizations to deliver clean and high-quality code, by conducting static code analysis to identify code smells, bugs, and security vulnerabilities. With support for over 30 programming languages, like Java, C#, JavaScript, or Python, SonarQube ensures that code is thoroughly analyzed against various coding standards, enabling developers to identify and fix issues related to code quality, bugs, security vulnerabilities, and code coverage. One of its key capabilities is seamless integration into the software development lifecycle, allowing for automated code analysis with each code commit, ensuring code quality throughout the development process. Allied to this is the comprehensive dashboard and reporting capabilities, helping developers to easily track the progress of code quality metrics, gain insights into areas that require improvement, and make informed decisions to enhance code quality and maintainability. Additionally, SonarQube offers API access controls/permissions, activity tracking, enable further customization and automation, and bug tracking, which contribute to efficient collaboration and project management. It supports a wide array of plugins and integrates with popular CI/CD tools like Jenkins, Azure DevOps, and GitLab, enhancing its adaptability and utility [9]. This integration streamlines the workflow for developers, allowing them to run code quality checks and receive feedback directly within their development environments.

This tool is scalable, suitable for small to enterprise-scale projects, making it adaptable for various development environments. Overall, SonarQube is a powerful tool for maintaining code quality and security throughout the software development lifecycle, despite

some resource requirements and a learning curve, offering a wide range of solutions and tools.

2.6 Pipeline Scans

A pipeline is a process that drives software development through a path of building, testing, and deploying code, also known as CI/CD (continuous integration/development), shown in Figure 2.3. It is a series of steps that must be performed to deliver a new version of software. The main objective is to minimize human error and maintain a consistent process for how software is released, by automating the process. Tools that are included in the pipeline could include compiling code, unit tests, code analysis, security, and binaries creation. Although it is possible to manually execute each of the steps of a CI/CD pipeline, the true value of CI/CD pipelines is realized through automation [4].

[10]

Figure 2.3: Development process with CI/CD.

The Pipeline Scan directly embeds into team development pipelines and provides fast feedback on flaws introduced on new commits. The Pipeline Scans evaluate the application at each commit, before the application is released into production environments, and by comparing the reports from each commit, it is possible to understand new vulnerabilities or errors introduced and in which commit.

Changes in results can be evaluated in comparison to previous scans, allowing to identify security flaws present in the application before releasing into production environments. Pipeline Scan provides the ability to establish a baseline of known security results. These results are stored as a JSON file called a "baseline" or baseline file. Pipeline Scan can compare the discovered findings with the baseline file to identify new findings. During the scan, Pipeline Scan ignores the findings in a baseline file and only uses the file to identify new findings [39].

2.7 Keycloak

Keycloak [11] is an open-source Identity and Access Management (IAM) solution that allows easy implementation of single sign-on for web applications and APIs. This tool is used to manage access to applications and interfaces (API) in a simpler way. Users authenticate with Keycloak rather than individual applications. This means that the

applications do not have to deal with login forms, authenticating users, and storing users, making the implementation easier for the development team. Enabling login with social networks is obtained through the admin console, just by selecting the social network to add. No code or changes to the application are required [**Keycloak**]. The fact that it is lightweight, fast, scalable and that it has a console that allows the tool to be easily configured, makes it a valuable help. This management is obtained through a subdivision of elements. The tool defines groups, roles, credentials, and a set of users, which are managed by realms. A user belongs to and logs into a realm, which is isolated from other realms and can only manage and authenticate the users that they control.

Clients are entities that can request Keycloak to authenticate a user. Most often, clients are applications and services that want to use Keycloak to secure themselves and provide a single sign-on solution. When a client is registered, the worker must define protocol mappers and role scope mappings for that client [12].

Keycloak presents a wide range of possible configurations, which makes it complete, but at the same time complex to use. Depending on the version, it can be easier or more difficult to make a small mistake that leaves a vulnerability in the tool's configuration, which can be used to attack the system. If a thorough and detailed configuration is done, it is possible to obtain a secure configuration.

2.8 KONG

Kong [13] is a scalable, open-source API Platform (also known as an API Gateway or API Middleware). Kong was originally built by Kong Inc. to secure, manage, and extend over 15,000 microservices for its API Marketplace, which generates billions of requests per month. Under active development, Kong is now used in production at hundreds of organizations from startups to large enterprises and governments including: The New York Times, Expedia, Healthcare.gov, The Guardian, The University of Auckland, Ferrari, Cisco, SkyScanner, Yahoo! Japan, Giphy and so on [13]. Postman API Platform [14] is an alternative to Kong, although it has a better grade on Gartner, there are a lot of issues related with incomplete fault reports, bad documentation/support, software failures and so on [57].

The primary function of the API gateway is to provide a single, consistent entry point for multiple APIs, regardless of how they are implemented or deployed at the backend. An API gateway takes all API calls from clients, then routes them to the appropriate microservice with request routing, composition, and protocol translation, typically, by invoking multiple microservices and aggregating the results, to determine the best path [15]. The Kong API Gateway is a tool that has a limited but free version, but also an Enterprise solution, which allows the adjustment of the price to the necessary features and not pay for what is not used. It is possible to obtain a Kong GUI through Konga [78]. Konga lets users run Kong's functions graphically, however, there have been no updates since 2020, which may raise some security questions.

Kong, unlike Keycloak, is a tool that does not need as much care when configuring it, since it does not have as many flaws in the default as Keycloak. Later in the document, we will discuss the known flaws of each of these tools and how to mitigate them to achieve the most secure configuration possible.

2.9 Spring

Spring [16] is a Java framework created with the aim of facilitating the development of applications, exploring, for this, the concepts of Inversion of Control and Dependency Injection. Spring provides resources for most applications, such as modules for data persistence, integration, security, testing and web development. Inversion of Control (IoC) allows users to delegate to another element (in Spring is named Container) the control over how and when an object should be created and when a method should be executed, for example. Thus, these responsibilities are now managed by this element, and it is no longer the responsibility of the programmers.

Dependency Injection (DI) is one of the ways to implement Inversion of Control. With Dependency Injection, the class no longer has to worry about how to resolve its dependencies. And that leads to one of the best-known features when programming with Spring: it is not needed to use "new" to create the objects managed by it, as this is done by the framework [17]. Spring Boot is designed to get the applications up and running as quickly as possible, with minimal upfront configuration, with the possibility of programming in Java, Kotlin, or Groovy. Spring's production-ready features (like tracing, metrics, and health status) provide developers with deep insight into their applications. Regarding security, Spring Security supports many industry-standard authentication protocols, including SAML, OAuth, and LDAP, and gives protection against top OWASP attacks [16].

Later, we will again address the topic of flaws associated with the various tools and parts of Link's reference architecture, in more detail, however, for now, we will only address the more general issues in the field of security of these tools.

2.10 Warnings NG

Warnings Next Generation (WNG) [88] is a Jenkins plugin specifically designed to provide users with a powerful and comprehensive tool for managing software project warnings. The plugin integrates seamlessly with other Jenkins plugins, such as Code Coverage and Static Analysis tools, making it possible to gather the results of tests carried out by various solutions. It offers a flexible and powerful set of rule-based filters, as well as an intuitive dashboard for quickly reviewing warnings and taking corrective actions [87]. It also offers advanced options for grouping and sorting warnings, and for setting thresholds and limits. Additionally, one of the most notable features of the plugin is its customization options, allowing developers to tailor the plugin to their specific needs, and

to configure custom rules and notifications for different types of warnings. Moreover, by providing users with an intuitive dashboard and powerful rule-based filters, WNG makes it easier to quickly identify and address problematic warnings all in one place [88]. This integration tool enables developers to identify potential issues earlier in the development process, which can save time and resources. It can be used to help developers and DevOps teams manage and monitor software project warnings in a more efficient and organized manner. Additionally, the plugin's easy-to-use interface and clear reporting make it easy for developers to identify and address issues quickly.

As well as being able to collect and analyse warnings from dozens of tools, it also offers the possibility of defining a parser so that even more tools can be added. If this feature is not suitable, it is also possible to analyse the expected format and generate a report in the same format, so users can see the warnings generated by tools that have not yet been declared [18]. Given this information and having analysed the features and functionalities, it was considered that this would be a very useful tool for providing an interface so that warnings don't just appear in a file or on the console. In this way, it will not only be possible to analyse existing vulnerabilities in a more intuitive and simple way, but also to filter and group data so that vulnerabilities can be fixed as quickly as possible.

2.11 Conclusions

Keycloak is a very complete, safe, and reliable tool, however, as with everything, there are exceptions and errors that can compromise the security we expect from Keycloak. There are a lot of settings that can be adapted when starting and defining Keycloak. These settings must be well analysed and studied before launching the service, because the tool has some known vulnerabilities, some mitigated between version updates, others persisted. These configuration errors are not seen as failures or insecurities of the tool, as passing failures and gaps to the launch phase is the responsibility of the programmer responsible for defining the Keycloak instance. That is, the flaws and errors that exist in the Keycloak launch phase in a production environment result from a bad configuration, or incorrect configuration, of the Keycloak instance.

This misconfiguration, or error, on the part of the programmer is easily present, since Keycloak's default configurations allow for some security flaws. However, a correct and meticulous configuration can mitigate these failures and generating a safe solution. It is, however, easy for the programmer to miss something, leaving a possible gap that could be taken advantage of by someone with bad intentions, which is why it is proposed that the Keycloak settings be checked periodically, automatically, so as not to interfere with the normal functioning of the tool in a production environment, but still guarantee the safety of the product. One way to do this configuration check is using pipeline scans. By resorting to pipeline scans, we will be able to verify the settings without bringing extra effort to the pipeline, and if changes are made to Keycloak, not only is the new version tested, but the results are also compared with the previous version, and it is possible to

get the differences and understand what changes were made between versions and what changes brought to the security of the instance.

Regarding Kong, the problems reported, apart from being few in number, arise from vulnerabilities or flaws in its source code, which will require changes to be made by the tool's developers in order to correct them. Unlike Keycloak, as with other tools that will be mentioned in the future, the solution will be version updates, which, given the number of vulnerabilities declared to date, will not be that frequent.

That said, using this method, the number of existing vulnerabilities has already been greatly reduced and the application is safer than ever. However, other questions arise: Let's imagine that attackers, users who hope to damage or extract something from the application, try to hack Keycloak and fail, try to hack Kong, and fail. Now, it would be expected that the application would be safe, however, imagine that the attackers manage to obtain the source code of the application in development (the product being secured by Keycloak and Kong), or part of it. This is a worrying situation, as programmers often end up writing code with the idea that no one will read it, thus leaving some information in the code that they would not leave if they remembered that the code could be accessed. Moving on to a more concrete example, let's imagine that the programmer explicitly inserted the link to access the database in the code. If anyone has access to the code, they will see the link to access the database and will now be able to make direct requests without going through the rest of the application's structure. Therefore, the source code must be analysed from the point of view of quality and security. There are several ways to do this, as seen previously, however, since what is intended is to obtain an analysis of the quality and security of the written code, it is considered that static analysis techniques (SAST) are the most adequate to do this. Ensuring good configuration of Keycloak and Kong, as well as quality and secure code, is a giant step towards ensuring application security.

3

Related Work

Before moving on, it is important to review the situation. We addressed the SAST methodology, which by analysing the code and creating tests that check for potentially fragile situations, allows users to check the application for vulnerabilities that could affect its security. In this case, testing is performed to verify situations such as SQL Injection, or others that may arise from code with bad practices, or just human failures. Then we analysed the DAST, which, this time without access to the code, tries to discover flaws in the applications, through, for example, the insertion of SQL code in fields that expect other data. By analysing the response of the web application, as well as the errors obtained, it is possible to identify some flaws in the code.

We then moved on to IAST, which monitors applications, and offers some advantages such as low execution times and still does not add any extra time to CI/CD pipeline. We also saw RASP, which allows users to monitor and correct threats to web applications in real time. However, these are focused on application security, since that is the focus of the work, but there are other aspects that can be addressed, such as load or stress, for example. By analysing the response times, depending on the number of requests, it is possible to monitor the application's capacity and predict possible future failures. Monitoring can also be done based on the CI/CD pipeline, in which new versions are checked based on the results of previous versions, in order to detect problems that may have been introduced in this new version. External security testing tools can also be used, which automatically perform the steps normally performed by intruders, in order to identify possible flaws in web applications.

These tools, metrics and methodologies are very useful and fundamental to guarantee the quality of the final product. Without some of them, it would not be possible to reach the vulnerabilities associated with each tool present in Link's reference architecture, mentioned above. It is on them that we will now turn to the next chapters. We will go deeper into this subject, what are the known vulnerabilities and how we can mitigate them to obtain a safer final product.

3.1 Keycloak vulnerabilities

Some of the following vulnerabilities were found in version 15.0.2 of Keycloak, so it is not guaranteed that these vulnerabilities have not been corrected in subsequent versions, since, at the time of development of this document, version 20.0.2 is available. Despite the possibility of correction in later versions, it was still considered important to mention these vulnerabilities, as the latest version is not used by all companies, thus being able to incur some vulnerabilities still present in the versions in use.

As discussed earlier, a realm manages a set of users, credentials, roles, and groups. These realms can be easily enumerated. To do so, just use a dictionary and brute force attack (repeated sending of requests by changing just a few parameters and analysing the response obtained and the execution times). In this way it is possible to obtain the existing realms; in the same way it is still possible to obtain the existing users in a given realm. The Keycloak Brute Force Attack detection does not prevent this as it only locks out a user for a given username, since the username is changing, this cannot prevent these attacks. [58] This kind of detection may prevent an intruder from brute forcing the password, since the user would be the same, although, using prevention based on the IP address of the sender instead of the username of the request would be much more efficient, since all the requests would come from the same IP address, regardless of the username.

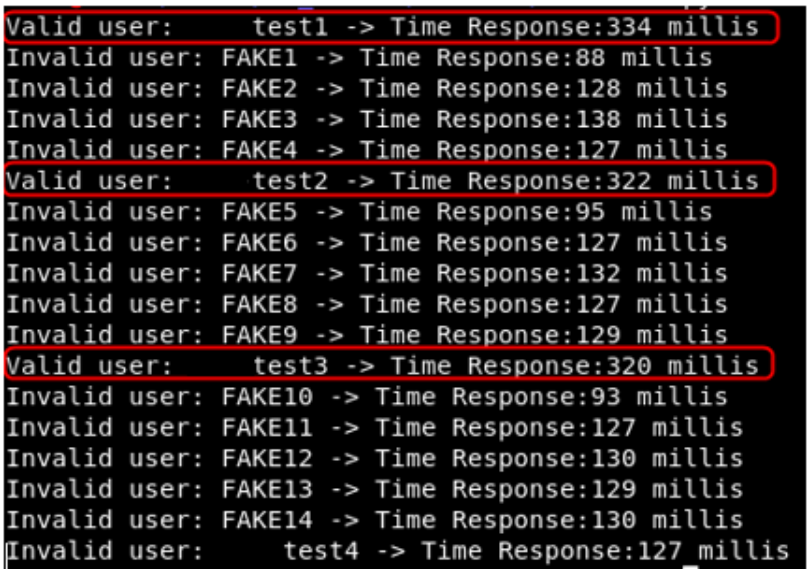


Figure 3.1: System response from different requests, only changing the username.

As can be seen from Figure 3.1, when a valid user is received, the response time is about three times the response time obtained when using an invalid user. In this way, it is possible to see which users exist (valid) or which do not exist (invalid).

Realms can be configured to allow user self-registration. Just in case the client is using a custom template for the login page, hiding the registration link, we can still try to directly access the registration link. This would let an intruder register on the realm, so,

in production environment, it is recommended disabling the self-registration.

It is also possible to enumerate valid email addresses from an authenticated perspective via Keycloak's account page. When changing the email address to an already existing value, the system will return 409 Conflict. If the email is not in use, the system will return '204 - No Content'. This vulnerability may be fixed by enabling Email Verification, this way will be sent out a confirmation email to all email addresses tested [81].

Keycloak can be downloaded by zip, or GitHub. Considering GitHub as the preferred choice for downloading Keycloak, version 15.0.2 was the most downloaded version (approximately 164k downloads). The following versions did not get as many downloads, with versions 16 and 18 getting around 100,000 downloads [73]. For this reason, the vulnerabilities presented will be identified in versions above 15.0.2. This case, despite being present on versions from 12.0.0 to 15.1.1, was considered a situation that deserved attention due to the possible damage caused by an intruder. A vulnerability (CVE-2021-4133)[1] was detected that allows an attacker with any existing user account to create new default user accounts via the administrative REST API even when new user registration is disabled. In most situations, the newly created user is the equivalent of a self-registered user and does not have the ability to receive any additional roles or groups. To carry out this attack, a user account is required, which, if self-registration is disabled, cannot be created by the intruder.

On version 17.0.0, the clients-registrations endpoint allows execution of JavaScript code on the client-side, which makes it vulnerable to a Cross-Site Scripting (XSS) attack (CVE-2021-20323) [82]. An intruder could use XSS to send a malicious script to a user, who, trusting that he is on the correct page, which he is, will trust that the script is trustworthy. Also, the end user's browser has no way to know that the script should not be trusted and will execute the script. Because of this, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser, and even rewrite the content of the HTML page [83]. This vulnerability is dangerous, as it allows obtaining a lot of information, some of which is necessary for user authentication. In other words, with this data the intruder will be able to impersonate the injured user, in the eyes of the entities crediting the credentials, allowing the intruder to operate on behalf of the user and perform the functions that the user can perform. Although fixed in version 18.0.0, in previous versions there was a privilege escalation flaw in the token exchange feature of Keycloak. Missing authorization allows a client application holding a valid access token to exchange tokens for any target client by passing the *client_ID* of the target. This could allow a client to gain unauthorized access to additional services [64]. In this way, the intruder can obtain more power than the one initially granted, which can cause much bigger problems for the application.

[1]The CVE-XXXX-XXXXX code characterizes the Common Vulnerability Exposure, and is unique to each vulnerability and allows distinction between the various publicly disclosed cybersecurity vulnerabilities existing in the software world. The database with all the vulnerabilities is managed by an accredited set of partners, responsible for checking the veracity of the detected weaknesses

In versions prior to 18.0.0 it is possible to find a vulnerability that allows arbitrary JavaScript to be uploaded for the SAML protocol mapper even if the *UPLOAD_SCRIPTS* feature is disabled. Security Assertion Markup Language (SAML) is an open standard that allows identity providers (IdP) to pass authorization credentials to service providers (SP). This means a user can use one set of credentials to log into many different websites. If an intruder manages to introduce code into the protocol mapper, he may have access to tokens or passwords, which allow him to authenticate himself to the protocol and impersonate the user. Since the protocol allows the use of the same credentials on multiple platforms, the required tokens will also be the same, so the intruder will be able to access the various portals connected to that protocol. In this way, the attacker will have more possible targets to try to attack, as the credentials do not vary between platforms, since they use SAML [68].

It is possible to see that Keycloak is a very useful and complete tool, despite the existence of some flaws. It is noticeable that, in most cases, security flaws originated from poorly defined variables, or not defined at all, so they can be mitigated through a meticulous configuration of some parameters and variables, as well as through the upgrade to a newer version (in some cases). If a proper configuration is done and the necessary measures have been taken, Keycloak is a very safe and reliable tool.

3.2 Code vulnerabilities

Code vulnerability is a term related to the security of software. It is a flaw in the code that creates a potential risk of compromising security, by opening a breach for attackers [37]. These flaws can be errors that go against good programming practices, or they can just be situations that escaped the programmer and that could be used to manipulate the application and obtain more information than what would be expected.

The Open Web Application Security Project (OWASP) is a non-profit foundation that works to improve the security of software. The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications [76]. Every year the list is updated, entering the most exploited vulnerabilities, and leaving those that are losing notoriety. In 2021, Broken Access Control rose from fifth to first place, as 94% of applications were tested for some form of broken access control. Access control enforces policies such that users cannot act outside their intended permissions, thus preventing users with minimal permissions from making compromising changes to the application. In case of failure, the possibility remains open for attackers to gain privileged access and thus gain control over the application [75]. This ascent proves precisely the need for the work developed in this dissertation, since Keycloak is precisely responsible for managing access. As mentioned, Keycloak has some Broken Access Control flaws, which will be mitigated at the end of the dissertation. The expected result will contribute to a reduction or even mitigation of

this type of vulnerabilities in applications that use Keycloak. To test this vulnerability, we may use BurpSuite or CrashTest Security, for example.

Cryptographic Failures moves to the second position in the table, which may lead to sensitive data exposure or system compromise. This failure can originate from the absence of cryptographic mechanisms, or from the use of deprecated mechanisms. In this way, attackers may have direct access to passwords or other important information, or even have easier access, if deprecated mechanisms are used. From first position to third, we have Injection, which consists of allowing users to enter non-validated or filtered information, in order to obtain something from the application.This type of vulnerabilities can be mitigated using tools such as TruffleHog, which allow users to analyse the code in search of hard-coded passwords, outdated encryption, etc. Within this section are vulnerabilities such as SQL Injection or Cross-site Scripting, which allow data that could compromise it to be passed to the application, and can be tested, also, using Burp Suite, for example.

In fourth place, we have Insecure Design, which refers to vulnerabilities that may exist in the application's architecture. There are still other vulnerabilities related to outdated components, authentication and identification failures, among others. Normally, these weaknesses are not purposefully created by programmers, they can result from distractions, underestimation of dangers, sloppiness, among others. Since they were not planned by the programmer, identifying them can be tricky and time-consuming, since some of the vulnerabilities arise from a set of factors, which can be difficult to identify in full.

A large part of the existing vulnerabilities in the code are simple errors, passwords, credentials, links, hard-coded, that is, explicitly written in the source code. This way, someone who has access to the code will have access to the credentials. Let's imagine the situation in which the application is protected with the correct mechanisms, however, somehow, the attacker gained access to the source code, where the link to access the database is inserted. The attacker, through the application, could not access the database, since the security was well planned, however, through the access link, he can have direct access to the database, without requests going through the application. In a situation where access data are inserted in the code, the attacker can obtain this data, if he has access to the source code, and can affect the application, in an authenticated way.

Hard coded passwords or private links, and bad variable usage, are examples of errors known as bad practices, that commonly escape the programmer and can result in vulnerabilities that can endanger web applications, so it is essential that vulnerability detection methods are used. Let's therefore analyse some vulnerability detection tools.

3.3 Testing Tools

Let's now analyse some existing solutions that can be useful to help solve existing problems in the company's reference architecture. We will start by analysing SAST tools, followed

by pipeline scan tools, since these will be the methodologies to be used in the next phase.

3.3.1 Static Application Security Testing

One of the problems with SAST tools is the production of many false positives, which end up discouraging programmers from actually existing errors. Some solutions are starting to integrate Artificial Intelligence to try to separate false positives from existing positives, and thus reduce the number of false positives.

Veracode is a company that offers many security-related software solutions. We will focus on SAST Veracode Static Analysis [19], which has a low false-positive count and offers developers potential solutions to issues it finds. It offers a variety of features including composition analysis, security management, audit trail and reporting. Being Software as a Service means low setup overhead and a quick turnaround from first acquiring access and getting results. The platform allows software developers to conduct application analysis and receive automated security feedback in the IDE and CI/CD pipeline. This tool, despite having many advantages, turns out to be a commercial tool, which requires some kind of cost on the part of those who use it. In addition, it is a cloud solution, which would require the code to be analysed to be transferred to the tool, something that in a commercial environment is not very secure [20]. For this reason, other similar tools, also in the cloud, were ignored. That said, despite being an interesting tool, the fact that it is a cloud solution, that it has costs, and that it has much more functionality than would be necessary, this solution will be discarded.

Checkmarx [21] is another existing tool in the field of vulnerability detection. This tool, in addition to supporting numerous languages right out of the box with no configuration, also offers SAST, SCA, DAST solutions, among others, to cover the needs of users. Despite being a very complete tool, it again suffers from the presence of many false positives. Although it has an outdated interface, compared to current standards, it is still a capable tool [61]. Once again, it is a solution that entails costs for the company, something that at the stage in which the company finds itself, is not a priority. For this reason, free solutions will be prioritized.

Moving now to the open-source tools business, Insider [40] is the open-source project of the Insider Application Security Team. This tool has its focus on covering the vulnerabilities referred to in the OWASP Top 10, analysing the source code, focused on an easy to implement software inside the DevOps pipeline. Currently, the tool supports JAVA, Kotlin, JavaScript, among others. There is also the possibility of verifying the code directly in the repository, integrating the tool together with the GitHub repository, for example. This is a relatively recent and free tool that can be an asset to ensure code security. Using it can be advantageous for static analysis of the code, but it is not enough to fix the problems related to Kong and Keycloak.

Finally, it is important to mention yet another tool, SonarQube [8]. This tool has support to analyse 29 programming languages (only 16 in Community Edition), integration with

platforms such as GitHub, Azure, BitBucket, as well as with CI/CD systems. This is a very complete tool, which allows, among many other functions, the creation of some minimum code quality requirements without which the code does not pass. This tool also ensures that no hard-coded keywords or links are introduced [22]. This tool is already used, on their own initiative, by some of the company's programmers, to ensure quality in the results produced. This tool is capable of fulfilling the necessary requirements regarding source code security, since it covers the possibilities of existing vulnerabilities. It features paid versions for programmers or companies, but it also offers a free version that allows code analysis throughout the development process, as well as code analysis as it is written, analysis of new pull requests, among other features, which allow this free version to fulfil the needs defined earlier in this document [23].

Having too many false positives continues to be a problem in most SAST tools today. Some tools try to mitigate this issue by resorting to artificial intelligence, however other tools, like Pulse-X [24], use logic, for example. An error detection possibility is through Incorrectness logic, used in tools such as Pulse-X. Where Hoare Logic proves that at most a set of states is reachable, we can use Incorrectness Logic to prove that a set of states is definitely reachable. While we can show correctness using Hoare Logic to demonstrate that a bug is not reachable, we can prove incorrectness using Incorrectness Logic by proving that a bug is certainly reachable [45]. Tools using logic of this kind are still scarce, however we have, for example, Infer, a logic-based analysis tool.

Infer [25], mentioned above, is a static program analyser for Java, C, and Objective-C. Infer is currently tracking for null pointer exceptions, resource leaks, annotation reachability, missing lock guards, and concurrency race conditions [25]. This tool is open source, can be used for free, and can be an asset to ensure greater code security, however, it is considered that its scope falls outside the proposed objectives, not bringing us much use.

These solutions usually come at a cost. This issue of cost is resolved with the emergence of open-source solutions, which allow maintenance by the community and therefore do not have costs. There are also solutions that analyse the code in a cloud environment, however, to do so, it would be necessary to transfer the code to the tool's environment, and code outside the company's environment is usually not something that pleases the company, for logical security reasons. However, tools such as SonarQube or Insider can be very useful and cost-free, making it possible to guarantee quality standards in the source code in a simple and economical way.

3.3.2 Pipeline scans

Enterprises are making their moves toward DevOps methodologies and Agile culture to accelerate delivery speed while ensuring product quality. In DevOps, what makes fast and reliable deliveries possible is the continuous and automated delivery cycle. This results in the need for proper continuous integration and continuous delivery (CI/CD)

tools [26]. Using the appropriate tools, it is possible not only to speed up the development process, but also to ensure automated verification and testing of the results obtained, in order to guarantee certain quality and safety standards. There are several tools available, including some previously mentioned in the SAST part (Veracode, for example) that have CI/CD integration capabilities, in order to automate some tests. However, there are tools dedicated to CI/CD, and it is on these that we will base ourselves. These tools bring several advantages, such as the need for smaller code modifications, isolation of faults, increased test reliability, easier updates and maintenance, among others.

CircleCI [27] is a tool that supports rapid software development and publishing. This tool can be integrated with GitHub, GitHub Enterprise, and Bitbucket to create builds when new code lines are committed. CircleCI also hosts continuous integration under the cloud-managed option (something that at an enterprise level might not be ideal) or runs behind a firewall on private infrastructure (this solution seems to be more suitable for companies that don't want to move their code). This tool also allows the creation of quick tests, continuous and branch-specific deployment, is highly customizable, allows fast setup and unlimited builds, automated parallelization. This tool is quite complete and widely used around the world, although it is a paid version, with no possibility of getting a free version, so for the company's needs there may be a more suitable and less expensive solution.

Another tool possibility is TeamCity [28], which is a JetBrains's build management and continuous integration server. TeamCity runs in a Java environment and integrates with Visual Studio and IDEs. The tool can be installed on both Windows and Linux servers and supports .NET and open-stack projects. This tool provides a new UI and native GitLab integration, while still supporting GitLab and Bitbucket server pull requests. TeamCity provides multiple ways to reuse settings and configurations from the parent project to the subproject, runs parallel builds simultaneously on different environments, is easy to customize, interact with, and extend the server, and finally, it has flexible user management, user roles assignment, sorting users into groups, different ways of user authentication, and a log with all user actions for transparency of all activities on the server. However, all these functionalities require a cost, something that can put some companies off. TeamCity offers a free version, but only in the cloud and quite limited in terms of performance and features available compared to paid versions. The only self-hosted server version has customizable features; however, it also has a cost, which can vary depending on the options selected.

We have already seen two well-known and used solutions that, however, present some obstacles already mentioned. These same obstacles affect most solutions, so we will now analyse an open-source solution, Jenkins [27]. Jenkins is a cross-platform, continuous integration tool used to continually build and test software applications, and has been a popular choice among Java developers [67]. It is a self-contained Java-based program with packages for Windows, macOS, and other Unix-like operating systems. With hundreds of plugins available, Jenkins supports building, deploying, and automating

software development projects. Jenkins is easy to install and upgrade on various OS, has a simple and user-friendly interface, a lot of available plugins, easy configuration, supports distributed builds with master-slave architecture, supports shells and Windows command execution in pre-build steps, among many other features that make this tool complete, capable, but free [67].

Although Jenkins is a tool with many users, it does not mean that it does not have weaknesses or complaints from them. Problems such as an outdated interface, unmaintained plugins, or even poor technical support, are pointed out to Jenkins, however, other positive points, such as the wide range of plugins that can be easily connected, being hosted internally, end up dividing users [46]. Some companies prefer to spend money on a commercial solution, with better support, maintenance and security, yet others don't mind giving up these issues to work with a free tool. There are several comparisons between other tools and Jenkins, [67, 52] , however they end up showing valid free options, but that do not cover all Jenkins points; or in paid solutions, which may also not cover all Jenkins points. Despite the negatives, Jenkins continues to have strong positives, which end up keeping Jenkins in wide use.

3.4 Vulnerability Assessment and Management

There are various ways of looking for vulnerabilities in a system, such as penetration tests. However, it is also possible to identify existing components and check them for vulnerabilities. This method will not find new vulnerabilities, it will only be based on vulnerabilities that are already known. On the other hand, for this reason, it is able to present its functionalities taking less time and requiring fewer resources, since no attacks or penetration attempts will be made, but rather an analysis. This method is normally applied to analysing components, identifying them and realising which components may have vulnerabilities, however, for checking instances, such as Keycloak, these tools will not detect the vulnerabilities that are expected, only in their components. Nevertheless, some solutions that could try to solve the problem, such as OpenVas and Nessus, will now be discussed.

3.4.1 Nessus

Nessus [69] is a powerful and widely used vulnerability scanning tool developed by Tenable Network Security. It serves as an indispensable asset for organisations seeking to assess their network security posture comprehensively. Nessus is designed to identify vulnerabilities and misconfigurations across a wide range of systems and applications. Nessus conducts network scans, identifying potential vulnerabilities in systems, services, and applications, supporting various protocols. It operates by sending various types of network packets to target systems and analysing the responses to determine potential vulnerabilities. In order to be able to carry out its work, a dedicated server or workstation is

needed, with sufficient computing power and storage capacity to perform scans efficiently. Regular updates are also needed, to ensure Nessus has knowledge about the latest vulnerabilities [85]. Since it is also a broad tool, analysing various components, it will make the associated database large, requiring a lot of storage space. What's more, it will run for much longer and much more often, as it is a larger tool. Although there are free and paid versions, this tool offers more than might be needed, requires more resources than wanted, and makes it a broader solution that is not as focused on what is desired. While Nessus may detect some misconfigurations or vulnerabilities related to the underlying infrastructure supporting Keycloak, it is not specifically designed to assess the security of the Keycloak instance itself. Keycloak's security largely relies on proper configuration and implementation of access controls, authentication methods, and user management. [69].

3.4.2 Vulnerability Assessment System

OpenVAS [53], short for Open Vulnerability Assessment System, is a powerful open source vulnerability management and analysis tool designed to help organisations identify and mitigate security vulnerabilities in their IT infrastructure [74]. OpenVAS excels at identifying vulnerabilities across a spectrum of assets, including network devices, servers and applications, through a combination of network analysis, deep scanning and comprehensive vulnerability checks. One of the outstanding features of OpenVAS is its extensive database of known vulnerabilities, which includes Common Vulnerabilities and Exposures (CVEs) and the OpenVAS Network Vulnerability Test (NVT) feed, allowing OpenVAS to map identified vulnerabilities to specific CVE identifiers [84]. As with the previous tool, in order to use OpenVAS effectively, organisations need a dedicated server or workstation with sufficient hardware resources to carry out the analyses [49]. Although it is open-source and free to use, this means spending dedicated resources on this tool, which ends up bringing costs which, given the DBs required, will be a considerable amount, which is why, for the situation in question, it is considered excessive [53].

In terms of its applicability for verifying Keycloak, OpenVAS, like Nessus, excels at assessing the security of the underlying infrastructure. However, when it comes to assessing the specific configurations and security of Keycloak - an identity and access management solution - specialised testing tools and methodologies are usually required. OpenVAS can be useful for assessing the security of the servers and network components that support Keycloak, but it may not provide a complete assessment of Keycloak itself, which focuses mainly on identity and access management. To summarise, OpenVAS is a robust open source vulnerability assessment tool with the ability to identify and manage vulnerabilities in a variety of IT assets. Although it can contribute to the security of systems that support Keycloak, it won't be able to cover the problems raised, so although it will be useful, it won't be a solution.

4

Proposed Work

Now that an introduction has been made and the needs to be guaranteed have been explained, it is time to analyse the work that was proposed, and the tasks described in the previous phase. After analysing Link's reference architecture, it was concluded that its security depended mainly on a set of tools in use. Take the example of Keycloak, an identity manager responsible for allowing access to some employees and restricting access to others. If this tool fails or is tampered with, there is a possibility that access will be allowed to those who shouldn't have it, and this problem could simply involve increasing the privileges of some employees, or even allowing outside users (hackers, for example) to pass through. It is therefore necessary to analyse the various tools belonging to the architecture and understand which components could cause problems and how it will be possible to mitigate these issues. Part of this work has already been carried out in the previous phase of the dissertation, in which it was concluded that there were a number of possible vulnerabilities that could jeopardize the security of the infrastructure. We will now have a look at the possible problems that exist in the reference architecture, as well as the solutions that could correct these issues. The reference architecture is divided into several parts, such as security, frontend, backend and databases. Since some of these components are more accessible than others, the security concern given to each will vary, i.e. a component that has direct access from outside the network will require greater concern with security and maintenance, since it will be one of the main possible entry points for hackers. If it is not possible to enter the network, the security of the network's internal components may not be so crucial. With this in mind, we'll look at Keycloak first.

4.1 Keycloak Verifier

To the moment, Mitre's vulnerability database [29] detects 93 vulnerabilities in the Keycloak tool. Although some belong to older versions of the tool, versions that have fallen into disuse or have been updated, the number of vulnerabilities detected to date is relatively high, raising questions about the tool's security and credibility. However, it should be understood that Keycloak is an Open Source tool and is therefore maintained by a

community, with no cost to use. Despite this, it is a competent and capable tool which, despite the large number of CVEs detected, continues to offer solutions to mitigate these vulnerabilities. The solution to most of these vulnerabilities found is to update the version in use, however, whether due to incompatibility of components or for another reason, updating is not always the ideal solution, so it may be possible to adopt alternative solutions. Knowing all this, and given the lack of solutions on the market, one of the aims of this dissertation was to create a tool to check Keycloak's security. This tool should focus on Keycloak, since it has a considerable number of possible vulnerabilities, and try to ensure that the instance in question is as secure as possible. Solutions such as JFrog DevOps-Native Security [41], or even CVE-check-tool [48], are more extensive solutions, which have disadvantages, such as: requiring more access to the system and its components, thus making them a possible gateway to the system as well; also some require a considerable amount of resources and may run a number of times more than would eventually be necessary; and these solutions may be paid services, which will bring a cost that may not justify the security they bring. Jenkins is a tool that, through the continuous delivery (CD) pipeline, allows the entire code deployment process to be automated, and this is how instances are created. Taking all this into account, it was concluded that the solution could be a plugin which, taking into account the Keycloak instance in question, would perform the necessary checks based on known vulnerabilities.

The expected solution is to produce a plugin that has sufficient access to analyse the configurations of the Keycloak instance, as well as the necessary components present in it, but without privileges that could make the plugin a way into the system. This plugin should also be able to obtain the existing vulnerabilities relating to Keycloak, check if they fit the instance in question and propose a solution, which, whenever possible, will involve a minimal change to the existing configuration, thus avoiding updating the version if there is any impediment. Since vulnerabilities in these tools don't appear every day, and once they do, it takes some time for a solution to be made official, there is no need for updates every 4 hours (as in the case of CVE-check-tool), nor every day, it is enough that, for example, the tool runs every 5 days. It is also hoped that the plugin will not require a large amount of resources, such as a DB with millions of lines, since it only focuses on Keycloak vulnerabilities. This plugin will be declared in a pipeline script in Jenkins, and will run automatically, periodically, presenting the results in an interface in Jenkins, obtained by using the Warnings NG plugin. In this way, it will be possible not only to visualize the vulnerabilities that may be found, but also to do some grouping and filtering of results that may be useful. The proposed plugin will analyse Keycloak and produce warnings, which will be converted into Warnings, which will then be forwarded and displayed in Warnings NG.

Although many vulnerabilities are presented, it should be borne in mind that these are vulnerabilities distributed across dozens of versions, which have been updated over time and many of which are already out of date, so it is necessary to make a selection of the vulnerabilities that really affect users, so version 15.0.2 was taken as a starting point,

as it is an intermediate version, and the one with the most downloads [54]. Now that the vulnerabilities have been reduced, and our focus is on the vulnerabilities that really affect users, it will be necessary to analyse them and see if there really are alternative solutions to updating the version. After investigating, we concluded that some of the vulnerabilities could be mitigated by making small changes to the settings of the Keycloak instance. It will then be necessary to understand how to verify the existence of the vulnerabilities and put together alternative solutions to the version update. Although the number of vulnerabilities found after filtering is small, and given that it is a widely used tool, it is still considered appropriate and useful to produce the Keycloak verification plugin. Above all, in addition to producing alternative solutions, it will also be a monitoring and alert tool, when new CVEs are detected that affect the Keycloak version in question, thus mitigating not only the risks of already known vulnerabilities, but also alerting future vulnerabilities as soon as possible.

This plugin, if it proves successful, will be a useful tool for Keycloak users, of whom there are a considerable number, and who could also suffer from Keycloak vulnerabilities. Next, moving on to authentication and security, we have the Backend services, which are managed by the Kong API Gateway, responsible for routing requests to the intended services. If an attacker manages to get past Keycloak, through some vulnerability, they will be able to reach Kong, in which case they will be prevented from reaching certain services and performing certain actions. Or maybe they won't, since Kong, like any computer tool, can have vulnerabilities, which usually go unnoticed until they are disclosed. Although fewer, Kong also has some vulnerabilities. After a simple search, it is possible to see that there are few vulnerabilities, however, since the search is done by character matching, you get vulnerabilities that belong to other entities, such as Konga (an unofficial Kong interface). After filtering, the vulnerabilities are reduced, but these few remaining vulnerabilities still represent possible threats that need to be assessed. After in-depth research, it was possible to see that the existing vulnerabilities relating to Kong were scattered, with not really a common problem focus, such as some variable that can be changed, but rather specific components that have been updated to correct these issues, or some issue that has been corrected with a change in the code. For these vulnerabilities, there is really nothing else to do but update the version, since the problem was present in the application and not in the configuration of the instance itself.

Having analysed these components of the reference architecture, it is then necessary to analyse the rest of the architecture in order to try to guarantee its security. Elastic Search is an AI-powered product, suitable for database search, enterprise system offloading, ecommerce, customer support, workplace content or websites, for example.This tool has some vulnerabilities in some versions. These vulnerabilities can be remedied by a version update. After a more in-depth investigation, it was possible to see that there were two possibly risky situations, since the instances in question fit the description of vulnerabilities CVE-2022-38779 and CVE-2023-31414 [72, 71, 63, 62]. The possible vulnerability had to be corrected by updating the version of the components in question. Apart from these two,

no other vulnerabilities were found. As with other tools, the problems found to date have no possible fix, apart from a version update, to a version whose function is to mitigate this vulnerability, so further investigation will not be of great benefit, since history shows a past with few declared vulnerabilities and with a common solution: updating the version in question. Continuing with the research, there is also a set of tools that are an integral part of the architecture, and which could possibly bring insecurity to the system in the event of a failure or defect, so they should also be analysed. These components range from integral parts of observability, monitoring, the frontend or even databases, some of which are more exposed to possible intruders, others less, either because of their function or their location, but all of which have security mechanisms and concerns. After analysis, it was possible to conclude that the version in use in the Redis component should also be updated, since the possible presence of vulnerabilities that could cause DoS, for example, was detected [30, 31, 32]. This issue, like the previous ones, can be solved by updating the version, since the problem arises from a flaw or distraction at source code level, so other changes would not correct the problem, only camouflage and try to hide it, which is not the appropriate solution. However, after analysing it, it was possible to see that the peer-to-peer (P2P) worm, [77] that recently appeared affecting Redis, does not fit into the architecture either and does not affect the Redis instance in use.

After analysing the rest of the components, it was possible to see that, to date, there were no more declared vulnerabilities that could fit into Link's situation, since the versions of the rest of the components had already been updated to problem-free versions. Taking into account the number of vulnerabilities, their nature and the frequency with which they have appeared over time, it was considered that it would not be beneficial to create more tools to analyse the rest of the components, since the number of vulnerabilities appearing over time has been low and the solution to these vulnerabilities has most often, if not always, been a version update. There are solutions on the market whose purpose is to carry out this analysis based on CVE databases and in certain situations they can be very useful, however, in the situation in question, since the number of vulnerabilities found was low, and based on previous years, it is not expected that there will be an emergence of vulnerabilities that will weaken the architecture. For this reason, and after all this research, it was concluded that the architecture was secure, and that with access to the tool produced (Keycloak Verifier), as well as some research from time to time, it is possible to keep the architecture as secure as possible.

4.2 Vulnerabilities in the reference architecture

After the research carried out earlier, it was possible to understand what possible vulnerabilities are present in the architecture, how to correct them and what the future plans are. In this chapter, we will analyse issues such as the tendency for new vulnerabilities to appear and what approach should be taken, for example. It will also explain why the aim of the dissertation has been achieved once the proposed measures have been taken.

The graph in the Figure 4.1 shows the distribution of declared vulnerabilities per tool over several years, so that we can conclude something about the evolution of the security of the various tools. For the sake of data standardisation, and so that there are no blank entries, 2018 was taken as the starting point. A few components of the architecture were selected, including the most important ones and those that could be more compromising, or others considered important after the previous research, so as not to compare a huge set of tools, and also so as not to expose all the tools in use by the company. The data in the graph was taken from the VulDB platform, which allows the calculation of the number of vulnerabilities that have appeared per year, among other categorisations. For now, for the purpose we need, the number of vulnerabilities per year is enough to try to understand some issues.

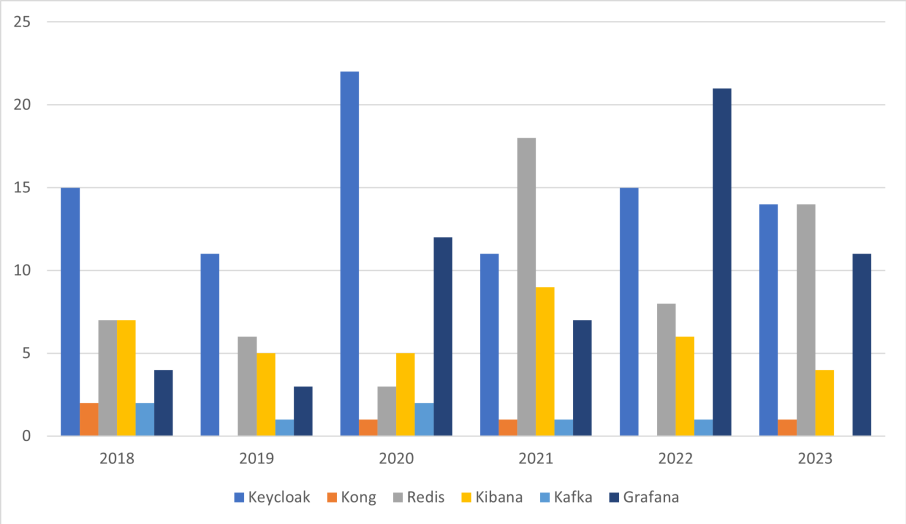


Figure 4.1: Distribution of vulnerabilities by tool, by year, based on databases of CVE.

Since 2018, Keycloak has had an average number of vulnerabilities of around 15, which, when compared to the rest of the tools, makes it the second most vulnerable tool. As this is an Identity and Access Management (IAM) tool, responsible for managing access to services, it is usually important to keep an eye on it. As it also has a wide range of possible configurations, it can easily be exposed by a misconfiguration. The fact that it is open-source also has an influence, as it may not have the resources of a costly tool. All these issues have resulted in vulnerabilities with solutions that can range from a change in the instance's configuration to a version update. Knowing this, taking into account the number of vulnerabilities that have arisen and their solutions, it is concluded that a plugin to check the security of Keycloak, based on CVE's, would be ideal, thus making it possible to guarantee with a little more certainty the security of the Keycloak instance, and consequently, the security of the company's architecture.

Kong, on the other hand, stands out for its low number of vulnerabilities, with an average of around 1 vulnerability per year (since 2018). In addition, the vulnerabilities were fixed with version updates, and there were no alternative solutions, since the problem

came from the source code, or from a component used in it, leaving users little choice but to update the version or accept the presence of the vulnerability. This issue led to the initial idea of creating a plugin similar to the one used for Keycloak being abandoned, since, apart from the small number of existing vulnerabilities, there were no solutions to propose other than the well-known version update. Looking at the data from Figure 4.1, you can see that there is a tendency for the number of vulnerabilities to emerge over the next few years to be minimal or zero, making it, based on the data, a very safe and reliable tool that does not pose any great danger or concern for the company's architecture. Redis has an average of 9 vulnerabilities per year, which is still a considerable amount, so it can be insecure. Once again, this tool, like others, tends to have vulnerabilities in the application's source code, so changes are made in updated versions to correct the problem. Since it is an open-source tool, it is to be expected that more vulnerabilities will be found, however, the instances are not accessible from outside the network, which drastically reduces the chances of an attack from the outside. It will, however, need to be reviewed from time to time to check compliance with possible new vulnerabilities, but once the aforementioned vulnerabilities have been found and corrected, there is no need for further concern.

As for Kibana [33], it is a tool that has presented an average of around 6 vulnerabilities per year, also resulting in resolutions to update the version in use, once again due to the source of the problems being the code itself. After vulnerabilities were detected affecting the instances in use, the solution proposed by the manufacturers was used, updating the version, which is then considered safe until further vulnerabilities are found. Taking all these issues into account and also the fact that it is not exposed outside the network, it was considered that there was no reason for greater concern about this component, since history dictates that it is a relatively secure component, as can be seen from the graph. Kafka, like the tools described above, has a low number of vulnerabilities, once again relating to the code, resulting in solutions that are, once again, software updates, so it is also considered that there is no need for extra concern with this component, as it is quite secure, given the low number of vulnerabilities [44].

Grafana [34] an open-source platform for monitoring and observability, has so far averaged around 10 vulnerabilities per year. Contributing to this is also the fact that it is open-source, as you may have fewer financial resources to invest in security research. Once again, the problems arise from the code or components used in it, causing versions to be made available to correct these vulnerabilities. To date, the version in use has no known vulnerabilities, and taking into account the emergence of vulnerabilities and their cause, it is expected that, even if vulnerabilities arise, they will be mitigated by updating the version in use, so it is considered that a periodic analysis, from a security point of view, will be enough to keep the instance safe.

Taking stock of the situation, it was realised that Keycloak was the component most likely to have vulnerabilities affecting the instances in use, possible vulnerabilities were discovered in other components (such as Redis), and it was also concluded that the rest of

the components were, to date, secure. It was also possible to clarify that the vulnerabilities found in Keycloak could arise more often from incorrect configurations, rather than the other vulnerabilities found, which are not only old, but also current and present in the architecture, the solution to which is a version update. The objective of creating a plugin to check Keycloak was also presented, which will carry out the necessary checks and suggest corrections or warn if any vulnerabilities are found. Ensuring the security of Keycloak already greatly increases confidence in the architecture. If, together with the plugin, periodic revisions are made to the versions of the components in use, of which there are not that many, it will be possible to guarantee the security of the architecture with greater certainty.

There are a few possibilities for checking components against known CVEs, such as OpenVas (Open Vulnerability Assessment Scanner) [74], or Vuls [86], but their main focus is not so much on checking this type of component (Keycloak, Kong, etc.), but rather on checking components that are more related to the code, so they don't fulfil all the requirements. Although they don't cost much, there would always be a cost and security could be compromised, since these are open-source components, so the cost is lower, with several contributors and regular updates, making it more likely that problems will be introduced to the network. This problem could be corrected by creating a plugin that, like the Keycloak plugin, runs periodically and checks the versions of the various components for any known vulnerabilities. However, such a plugin would have to have one of two options: privileged access to the system's components or receive the components and versions as parameters. The first option would result in the plugin having very high access, which could put the architecture in even greater danger, since once the plugin is compromised, it will be possible to access all the components, which could be more catastrophic than the presence of a vulnerability.

The second option, despite being the safest, would mean that the work of collecting versions and components would still be done by an employee, and the automation of the process would be compromised, since the pipeline script would have to be updated when the version of a component changed, otherwise false security information could be passed on, since there would be a difference between the declared version and the one actually in use. In the second option, the plugin would remove the burden of researching the vulnerabilities of each component from an employee, however, considering the workload that is required, considering the existing options and available resources, and also considering the number of existing vulnerabilities and their tendency to appear, it was not considered that there would be a need to replace part of the work with a plugin. Although some effort would be taken away from collaborators by passing it on to a plugin, it wasn't considered to be such an effort that it would justify its creation. However, it is always possible that, if it is considered beneficial and useful, this plugin will be created in the future, since the basis for its creation may well be the proposed plugin (Keycloak Verifier), since it will do some of the same work, but with the difference of having a relatively lower complexity, without the need to search for component versions, as is

proposed in Keycloak Verifier.

In addition to its code analysis capabilities, SonarQube conducts a comprehensive battery of tests to ensure code quality and security. These tests encompass a wide range of aspects, including code maintainability, reliability, security, and performance. For instance, it thoroughly checks for code smells, such as duplicated code and overly complex functions, helping developers enhance code maintainability. Furthermore, SonarQube runs security tests to identify potential vulnerabilities like SQL injection, Cross-Site Scripting (XSS), and insecure authentication mechanisms. It also performs static code testing to catch bugs and issues early in the development process, reducing the need for post-release bug fixes. This multidimensional testing approach ensures that software projects not only meet coding standards but also align with best practices, ultimately resulting in more robust and secure applications. Having said that, it can be seen that the code has been checked and tested enough to be able to say with the greatest possible certainty that the infrastructure is secure, in terms of the code.

4.3 Conclusion

After analysing the existing vulnerabilities, their tendency to appear, the solutions proposed to date, the responses and adaptations of the tools to the vulnerabilities, the characteristics of the vulnerabilities and the characteristics of the tools in use, it is concluded that by using the proposed plugin periodically, accompanied by a review of the component versions in relation to known vulnerabilities, it will be possible to guarantee the security of the reference architecture with a little more certainty. Having understood the problems encountered and explained the solutions to be adopted a little better, we will now move on to a more detailed explanation of the work carried out at this stage, in order to obtain a plugin that can verify the security of Keycloak and thus try to ensure the security of the company's reference architecture with greater certainty.

5

Work done

This chapter discusses the plugin developed in this dissertation, as well as some of the problems encountered and how to overcome them. As explained earlier, the need arose to check Keycloak's security, and a search began for tools that could resolve the issue. Some possibilities emerged, but sometimes they cost more, or required resources that were considered unnecessary, since the solutions found maintained databases with the vulnerabilities, and also required them to be updated at relatively short intervals, which for the situation, and taking into account the data analysed above, was not justified. In addition, the solutions were options that needed to run all the time, which requires constant resources for something that will probably be needed maybe 2 or 3 times a month, increasing security costs unnecessarily. Let's now move on to explaining the plugin, so one can understand why it is a better alternative to the solutions we've found.

5.1 Keycloak Verifier

Starting at the beginning, Jenkins is currently used in the company, so that through pipelines there is coordination between code development, testing, etc. and code deployment, making the whole process easier. It would be interesting and useful if the plugin could run in the pipeline, to facilitate the whole deployment process and also automate the process of running it periodically. Allied to this was another question: what is the best way to create a customised plugin? The answer to these questions was Maven, basically a core framework for a collection of Maven Plugins, allowing the reuse of common build logic across multiple projects [66]. Using Maven, it is possible to automatically create the skeleton of a plugin of the selected format [50], which will run easily in Jenkins. Once the plugin has been created, it is time to gather the vulnerabilities in order to find their solutions and understand what to do about them. Since Keycloak has a wide range of versions, and after realising that version 15.0.2 was the one with the most downloads, we started from that version, gathering the vulnerabilities that affected that version, or more recent ones. This resulted in 16 vulnerabilities, and the search for solutions began. Since the number of vulnerabilities was not very large, it was decided to create a file that would

store all the vulnerabilities in a common format and that would allow them to be easily accessible afterwards, so that they could be executed even if it was not possible to connect to the source of the vulnerabilities. In addition, vulnerabilities could be easily altered or added to this file, thus making it possible to have a local DB, with a tiny fraction of the DB size that other alternatives required. After analysing the possible solutions, it was concluded, as expected, that the majority would be version updates, however there were vulnerabilities that did not apply in all cases, and there were also alternative solutions for a couple of vulnerabilities.

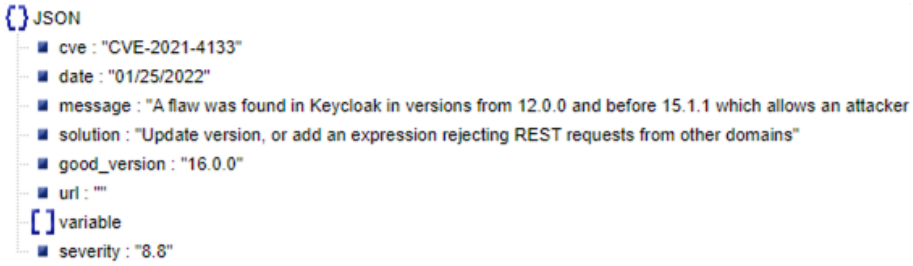


Figure 5.1: JSON object obtained from the CVE database.

A base model was then created, similar to a JSON object so that the attributes were easy to access, which has the fields shown in the Figure 5.1: CVE, date, message, solution, good_version, URL, variable and severity. The "CVE" variable is an integral part of all the objects declared in the file, as it is essential even for comparison with recent vulnerabilities. The "date" variable is quite enlightening and is present more for information purposes. The "message" is the original message that forms part of each CVE and will be presented to the user if a vulnerability is detected, explaining the vulnerability found. The "solution" will also be presented to users in order to give an indication that the vulnerability has been resolved. "good_version" represents the version from which the vulnerability does not apply, being an unaffected version as well. The "URL" variable is used in case there is a check available, this URL is consulted and in the response the value of the variables present in "variables" will be analysed. The "severity" is represented by a numerical value, part of the usual CVE DB scale. Now that we know what should be presented to users, we need to understand how this information can be passed on. In contact with other Link colleagues, a solution was presented that would fulfil this need. This solution is the Warnings NG plugin, which makes it possible to display the Warnings found, show distributions, graph the appearance of Warnings, among other categorisations.

This plugin has been designed to integrate with a wide range of code analysis tools, making it more geared towards analysing documents, which means that some Warnings NG's features cannot be used. Despite this, the plugin is prepared to accept other types of formats that may wish to use the plugin as an interface, which is exactly the situation we find ourselves in. After analysing the Keycloak instance, it would then be necessary to output the warnings in the expected format so that the plugin could read the results. This could be done in two ways: by creating a parser, or by using the original format of

the warnings. Creating a parser would be ideal, as it would allow the warnings to be "translated" from the format obtained by Keycloak Verifier into the format expected by Warnings NG. However, as we'll see later, some problems arose that made this impossible, so the second option was chosen. Now that we know that it will run in Jenkins and report in the format expected by Warnings NG, we need to understand how to detect the existing vulnerabilities.

Once understood what we are hoping to achieve, it is time to achieve it, and to do that we need the vulnerabilities. Starting with the vulnerabilities, keeping only a local database, although useful, does not fulfil all the needs, since if new vulnerabilities were to appear, it would not be possible to detect them, and to overcome this issue it would be necessary to obtain the existing vulnerabilities from some database. There are several tools that allow this, but they require some kind of payment, or they are mechanisms that basically replicate the database to local space, requiring millions of database entries to be present locally, which is not justified in this case. A public API then emerged that allows you to search by vendor or product, for example, in order to obtain only the vulnerabilities that really matter to us. After analysing the documentation, it was possible to understand the format of the request, after which it remained to understand the distribution of vulnerabilities by manufacturer/product. It was then concluded that "api.cvesearch.com/search?q=keycloak" would be enough to obtain the vulnerabilities that had ever been declared and referenced Keycloak in the affected products, or in the information message, but often the flaw was not in the Keycloak itself, it was only referenced in the information message. To filter out these false positives, further filtering was used, based on the product and manufacturer, allowing the vulnerabilities to be selected. The response obtained from the API request is a JSON object containing, in addition to some information such as the number of vulnerabilities found and of course the vulnerabilities themselves, associated information such as affected products, severity, etc.

As soon as only the vulnerabilities that matter have been obtained, it may then be possible to compare the vulnerabilities found with the vulnerabilities present in the local database, and thus detect the presence of new vulnerabilities. However, it is not yet possible to see if the vulnerabilities actually apply; for that, we will need to obtain the version in use. Since, for security reasons, this is only shown in the Realm information, there is a need for an authenticated user with permissions to access the Realm settings, however, too many permissions would make it a possible danger to the network as well, so it was necessary to select the necessary permissions. After analysing the existing documentation and the instance created for testing, it was possible to conclude that "view-clients" and "view-realm" would be the most basic configurations that would meet the needs, not posing any danger of altering data, since they are view-only permissions, and allowing the necessary fields to be viewed, fulfilling the security and requirements. These permissions will be associated with a user, who will be used in requests to the API, in order to obtain the necessary information. To do this, fields such as Username, Password

and client_ID are passed in the REST request to the API, which are necessary to obtain a token, used later in REST requests to the Keycloak API. These fields contain sensitive information which, although they don't allow much access due to limited permissions, could still pose a danger, so they are obtained from the plugin as environment variables, which are passed to it via Jenkins, where they are encrypted and stored. Jenkins also has a protection that does not allow credentials to be disclosed by displaying them in the console, i.e. a print command in the code will only print "****" instead of the credentials. The credentials can therefore be considered secure and are not considered a form of weakness for the architecture.

Once the vulnerabilities have been obtained, filtered and compared with the local database, it is possible to detect the presence of new vulnerabilities. If the vulnerability is present in the local database, it is this data that is presented to the user after the checks have been made, if applicable. However, if the vulnerability is not present in the local database, then we are dealing with an unknown vulnerability, in which case the warning is generated with the information from the database to which the REST request was made, thus allowing the vulnerability and related information to be displayed, even if it is not declared in the local DB. Once a known vulnerability has been detected, it may be that it depends on some specific configuration, in which case, in order not to give a false positive, the necessary check is made and if it is found to exist, the warning is produced. Since Warnings NG is not prepared for the reports generated by Keycloak Verifier, it is necessary to translate the results into a specific format. To do this, it will be necessary to select the variables to be displayed and map them to the variables expected by Warnings NG. Since it was impossible to add new fields, as will be seen below, it was necessary to adapt the required fields to the possible fields, and to do this some fields had to be deleted, such as the date the CVE was created. Even so, it will be possible to present details such as the original vulnerability message, the severity and the possible solution, which, although not exactly what was expected, is acceptable and fulfils the expected requirements. In this way, using Warnings NG it will be possible to present an interface for the possible vulnerabilities found, instead of just producing a result on the console. Keycloak Verifier, after obtaining the necessary configurations for its checks, after gathering the known vulnerabilities, will carry out its checks and produce a report. This report is a log-type file, containing JSON-type objects, with the vulnerabilities accompanied by the features that are allowed.

In the Figure 5.2 can be seen an extract of the result of running Keycloak Verifier in a corporate environment. May be seen two vulnerabilities found (for data protection reasons no further results will be shown), associated with the severity, an informative message and the proposed solution. It is also possible to see a distribution of vulnerabilities by severity, or even a distinction between new vulnerabilities and existing ones. The latter feature, in the Figure 5.2, only shows vulnerabilities as "outstanding", however, if any changes are made and a new vulnerability appears in the system, it will appear in the "new" category. Although these graphs are a visual aid, they don't provide much help, since we don't

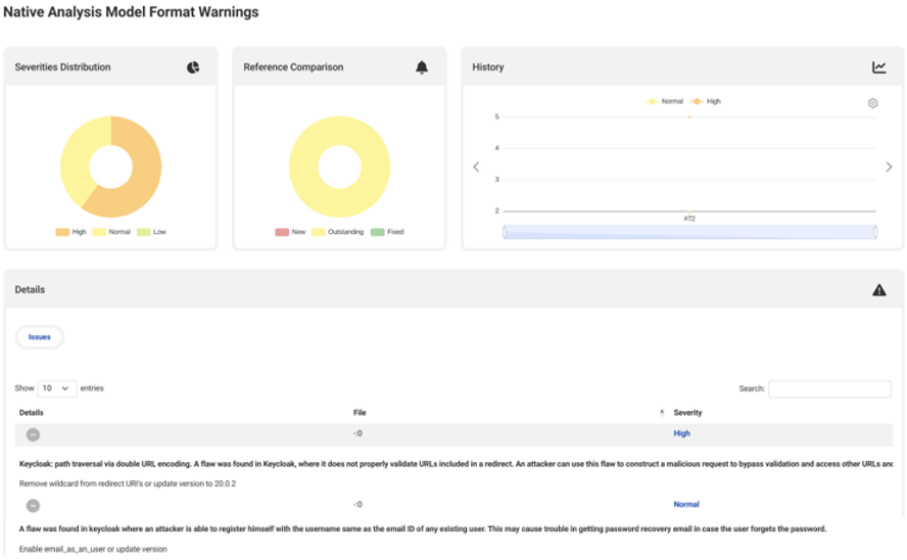


Figure 5.2: Warnings NG output after running the Keycloak Verifier in Jenkins.

expect many vulnerabilities to be displayed; however, the plugin allows us to see the vulnerabilities and their characteristics in a more appealing and easy-to-interpret way than just the console, which is why, although it was intended for SAST techniques, this plugin was used as an interface. This plugin will appear in a tab on the side of Jenkins, next to the other plugins in use, making it easy to access.

```
1  pipeline {
2      agent any
3
4      stages {
5          stage('Build') {
6              steps {
7                  // Get some code from a GitHub repository
8                  git url:                                    ', branch: 'main'
9                  withCredentials([usernamePassword(credentialsId: '            ',usernameVariable: 'USERNAME', passwordVariable: 'PASSWORD'),
10                 string(credentialsId: '    ', variable: 'HOST'),
11                 string(credentialsId: '      ', variable: 'CLIENT'),
12                 string(credentialsId: '      ', variable: 'API_KEY')]){
13                   withMaven(maven: 'Maven3') {
14                     sh 'mvn clean install'
15                     sh 'mvn org.myNisNine:keycloakVerifier-maven-pl gin:verifier -X'
16
17                 }
18
19                 }
20
21             }
22         }
23         stage('Analysis') {
24             steps {
25
26                 recordIssues( enabledForFailure: true, aggregatingResults: true, tools: [issues(pattern: '**/output.log', reportEncoding: 'UTF-8')] )
27
28             }
29         }
30     }
31
32 }
```

Figure 5.3: Example pipeline script to deploy the plugin.

To get the result present in 5.2, just run the plugin in Jenkins, using a pipeline script like the one shown in the Figure 6.2. The first line describes the pipeline, the fourth line describes the pipeline stages, the first being "Build", which will obtain the plugin from the GitHub URL shown (line 8), then from line 9 to line 12 the credentials needed by Keycloak Verifier are shown, which in this case are: the Keycloak Host, Client, Username and Password, so that the necessary requests can be made to Keycloak. In addition to these, an "*API_KEY*" is also presented, which is a key needed to make requests to the API of the vulnerability database, but we'll come back to this later. On line 13, the use of Maven in Jenkins begins, using "withMaven" to utilise Maven's functionalities in the

Jenkins pipeline [79, 80] and so, on line 14, it is possible to compile the plugin using Maven, followed by the command to run the plugin (line 15). This command is composed of "mvn", to use Maven, followed by the groupId of the plugin, followed by the artifactId and finally the goal to be executed in the plugin, which in this case is "verifier". In order to use Warnings NG, the "Analysis" stage was created, which is responsible for calling the plugin; using the "recordIssues" command, the pipeline recognises the plugin. The commands relating to the report generated by Keycloak Verifier are passed to it, in this case "issues" indicates that the report is in the native Warnings NG format, followed by the location of the report file and its encoding. In this way it is possible to capture the vulnerabilities (stage Build) and then produce an interface for the output (stage Analysis), automatically and with a periodicity that can also be defined, allowing a periodic passage through the plugin interface to detect the presence of vulnerabilities in Keycloak.

5.2 Problems Found During Development

Before starting the plugin development process, the existing documentation was analysed, and it was concluded that it was possible to convert the warnings to Warnings NG in two ways: a parser could be made, or the warnings could be converted to the plugin's basic format. The first option seemed the most appropriate, however, there are several possibilities for doing this, however, since it was easier, it was decided to define a Groovy parser in Jenkins [35], then it would only be necessary to call this parser in the pipeline script, and the warnings would be received in the correct format [36, 70]. This is because the parser would basically be a mapping between the values obtained by the Keycloak Verifier and the values expected by the Warning NG. However, it turned out that the parser only recognised the basic fields, such as "file", "message", basically the fields that were already expected, but it was not possible to add new fields. In order to remove the necessary values from the String received, the following regular expression was created:

{{([]+)}{([]+)}{([]+)}{([]+)}}

Which when intercepted with a String of type:

{{severity}{description is this}{solution is this}{category}}

You would get each of the fields individually, which would then be passed on to Warnings as follows:

builder.setMessage(matcher.group(2)).setFileName(matcher.group(1)).buildOptional()

In the example above, the result of the second group found ("description is this") is mapped to the warning message and so on until the groups are exhausted. However, since this tool is prepared to analyse according to SAST techniques, it has fields such as "file", which you can't remove, even if you don't assign any value, something will

always be displayed, even if it is "-:0". According to the plugin's documentation, it would also be possible to add extra fields using the "additionalProperties" property, but it was never possible to map this variable. Since it wasn't possible to do the parser correctly, we opted for another suggested way, which was to create the parser in Java, in the plugin, and pass it on with the plugin code, although this required the use of specific classes from "jenkinsci/analysis-model/../parser". These classes contained the implementation of the existing parsers and also allowed the use of some interfaces and classes to create a customised parser. However, this solution didn't work in the end, because although a parser was in fact created in Java, using the classes mentioned above, they required the use of a version of Java that clashed with other classes needed for the plugin, so this strategy had to be abandoned as well. So we tried mapping again using a groovy parser in Jenkins, the simplest solution, but after a long time trying to solve the problem, we tried running Warnings NG in its basic warnings format, using a file with vulnerabilities as input to Warnings NG, but this file is a test file, taken directly from the GitHub where the plugin is located.

```
{"fileName":"some.properties.text","severity":"NORMAL", "lineStart":10, "other":value, "additional": "stuff"}
{"fileName":"file.xml.txt","lineRanges":[{"start":11, "end":12, "other":10, "additionalProperties": "stuff"}}
```

Figure 5.4: Snippet of log file to be sent to Warnings NG.

The Figure 6.2 shows an extract from the file mentioned above. It contains some lines representing alternative ways of declaring the variables to be displayed by Warnings NG, but when I used this file, the result was the same: only a few fields were displayed, and it was impossible to define extra fields. Contrary to what the documentation says, the "additionalProperties" property could not be defined, giving an error and resulting in the output without these extra fields. Changes were made to the fields and their names, all the documentation for creating a parser, the Warnings NG documentation and various tests were analysed, but it was never possible to add extra fields as expected. Despite this inconvenience, it was possible to mould the information that was expected to be emitted into a format that could be read by the plugin, by using the basic Warnings NG format and emitting the warnings in that format, so there was no need to use a parser. The idea of using extra fields to define other properties, such as the date or CVE, has not been ruled out, and it is hoped that these flaws in Warnings NG will be corrected in the future, so that Keycloak Verifier can also be changed and show a more fit-for-purpose interface.

Once the previous steps had been completed, and the expected plugin had been obtained, it was then necessary to move on to testing the plugin. At this stage, another problem was encountered: a 403 error was constantly received, implying that there was an authentication failure in one of the requests executed by the plugin, which prevented it from working. After investigation, it was concluded that the problem came from the API used to obtain the existing vulnerabilities related to Keycloak. It seems that somewhere between the beginning and the end of the plugin's development, an API-KEY was introduced to restrict access to the DB [51]. Until now, this was the only completely

free solution for accessing an up-to-date vulnerability DB. There are alternative solutions, but as mentioned above, they come at a cost, or require a local instance, which entails other costs too, so we opted for the solution proposed by the API owners. This solution involved requesting an API Key, so an email was sent explaining the background to the problem, the context of this being a master's dissertation, and the response was positive, showing support and confidence in the project, so a Key with 3000 monthly accesses was made available. Since it is not considered necessary for the plugin to run on a daily basis, and since each run uses the API once, only 22 accesses would still be needed, which is more than enough for the key obtained. However, since the open-source solution is expected to be published, 3,000 accesses will not be enough to share with all users. If publication goes ahead, the possibility of obtaining a key with more accesses will still be discussed, or of accompanying users so that each one obtains their own key, which may even have fewer accesses, since the one currently obtained also includes room for testing. After introducing the API key into the plugin, the opportunity was taken to change the code a little, so that if it detects a flaw in access to the DB of the vulnerabilities, it will issue a warning, but it will work with the local DB, functioning correctly in it and still detecting the flaws, so it is possible to run the plugin without an API key. However, this last situation will result in the lack of detection of recent vulnerabilities that may not yet be part of the local DB.

5.3 Conclusion

Despite some unforeseen events and obstacles, it was still possible to successfully complete the task and obtain the desired plugin. This plugin, despite being able to run on Jenkins, also runs on any console that has Maven, producing a report that, despite being in the Warnings NG format, is very enlightening and easy to read, allowing you to obtain the same type of information (or more, since not all the fields have been mapped to the Warnings NG format) about the vulnerabilities, without requiring the creation of a pipeline script. It is therefore concluded that a versatile plugin has been obtained, with no need for large DB sites, no need to run regularly and no added costs, capable of analysing a Keycloak instance and detecting possible vulnerabilities and, where possible, suggesting an alternative solution. The work is not yet finished, as we intend to add new vulnerabilities to the local DB whenever new vulnerabilities appear, and we also intend, as soon as possible, to produce a more appealing and appropriate result in the Warnings NG interface. To this end, we will open the issue of the problem found with "additionalParameters", and as soon as it is resolved, the plugin will be improved, bringing a better experience for users, who are expected to be many, given Keycloak's reputation as a competent tool.

The tool is available temporarily at the URL "https://github.com/dredlops/testeFinal.git". Later, once the remaining corrections and improvements have been made, the plugin will be moved to the final repository, where it will then be available to keycloak users and

beyond.

6

Experimental evaluation

In this chapter, the tests will be carried out, and the results analysed to confirm the plugin's functionality. The tests were carried out using locally created instances, with versions 7.0.0, 20.0.0, 21.1.1 and 19.0.1 of Keycloak. The plugin will run on Jenkins and make requests to an instance of Keycloak. Both Jenkins and Keycloak will run on the Docker desktop, and in order for the REST requests to be sent correctly, the defined IP and host will be used during testing, without using the credentials stored in Jenkins. However, tests have been carried out in a company environment and the various applications have been integrated correctly, and the results obtained have been as expected, something that will be discussed in the future.

Starting with the oldest version, which, although it does not fall within the proposed range, we thought it would be interesting to include, so we'll start testing with version 7.0.0. This version has a vulnerability that causes a possible enumeration of emails, allowing access to emails that could be used to gain higher access to the system. This vulnerability can be resolved by deactivating the "account-console" and "account" clients, so it won't be possible to access the email exchange page and users won't be enumerated. In the Figure 6.1 can be seen the file with the output from Keycloak Verifier, where there is a large set of vulnerabilities, since the version in use is very old, however the selected vulnerability stands out. This is the vulnerability mentioned above, since the clients mentioned are enabled by default.

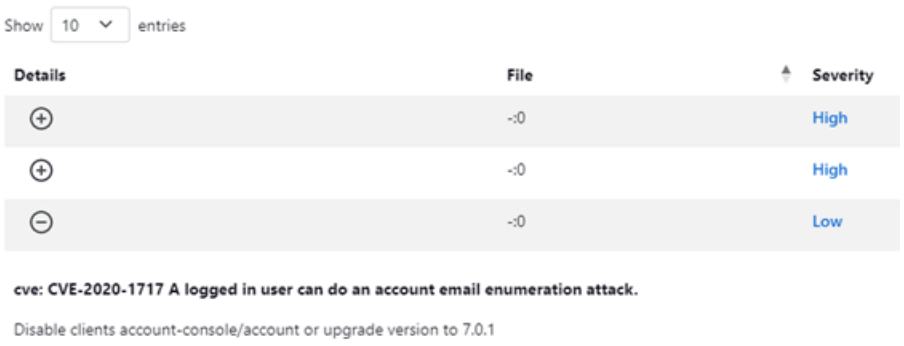


Figure 6.1: Warnings NG with CVE-2020-1717.

In the Figure 6.2 can be seen part of the settings for the "Account" client, in which can be seen that the client has the "enabled" variable set to "ON". By deactivating this variable (changing it to "OFF"), as well as the same variable in the other suggested client, you can correct this vulnerability.

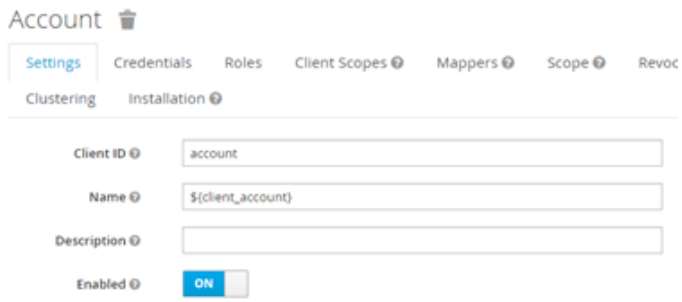


Figure 6.2: Account client settings.

In the Figure 6.3, can be seen the aforementioned result, showing the detection of the change in the variable, in this case, since the vulnerability no longer applies, the vulnerability is not emitted and does not appear in the output file for Warnings NG.

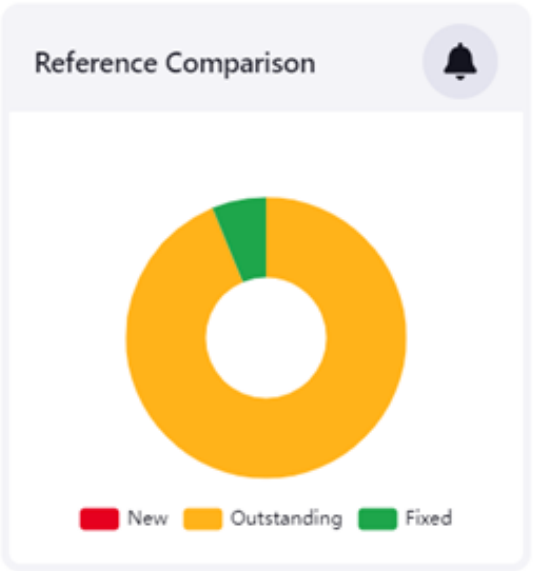


Figure 6.3: Warnings NG graph showing CVE-2020-1717 fixed.

This vulnerability, which was one of those that required specific concern, since it has an alternative solution to updating the version, after testing, its correct functioning is guaranteed, since the vulnerability is not only detected, but also verified so that a false alert is not issued, and only if it applies will it be notified. You can then move on to the next version that is part of a range of vulnerabilities. Another example of a vulnerability that requires confirmation and may have a workaround is CVE-2022-3782. This vulnerability can appear in versions prior to 20.0.2, so version 20.0.0 was selected to confirm this Keycloak Verifier check. In the Figure 6.4 can be seen the result of running the plugin,

accessing the Keycloak instance, version 20, emitting the vulnerabilities mentioned below, where can be seen the presence of the aforementioned CVE. After analysing the error message, if updating the version is not the appropriate solution, it is also suggested to remove the wildcards.

Figure 6.4: Warnings NG with CVE-2022-3782.

Analysing the Figure 6.5, looking at the "Valid redirect URIs" can be seen the presence of a wild-card, allowing all links that start with "/realms/master/account", possibly allowing an attacker to create a URL that passes this check and calls into question the security of the system. It is, however, possible to change the "Valid redirect URIs" so that they fulfil the requirements, while still limiting access to attackers. Once these situations have been changed and the wildcards removed, the vulnerability will no longer appear in the output file, as the vulnerability has been resolved.

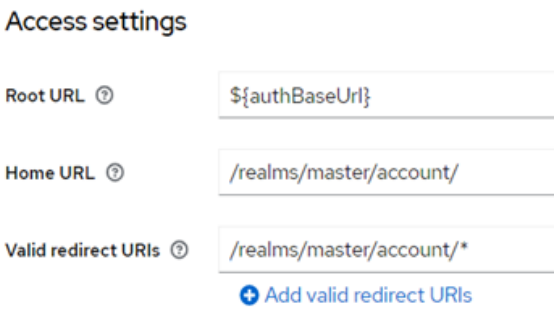


Figure 6.5: Example of a client's settings.

Some of the extremes of the affected version range have already been tested in order to reduce the number of false positives, but we will now test version 21.1.1, which according to DB should not present any vulnerabilities. Once an instance of this version had been created locally and configured, namely after creating the Keycloak user with the necessary access to the plugin, changing the necessary plugin variables and running the plugin, it was found that there was no vulnerability shown in the output file, proving that no vulnerability had been detected and that it was therefore, as expected, a secure instance. Having verified the security of the previous version, this time it was decided to test a vulnerability at the extreme end of the range, making it the last non-secure version, and after running the plugin, the information shown in the Figure 6.6 was obtained. The declared vulnerability was resolved in the next version, and it was not possible to resolve the problem in any other way than by updating the version. Since no other vulnerabilities affecting this version have yet been declared, no further vulnerabilities are produced, and after updating the version, the vulnerability will no longer be shown in the result.

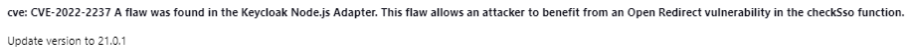


Figure 6.6: Output file from running Keycloak Verifier on Keycloak 21.0.0.

After checking the extremes and some other versions considered interesting, we will start testing the plugin in a corporate environment. To do this, a new pipeline was created, in which the commands for obtaining the plugin from GitHub, compiling and running it were introduced, as well as the creation of a stage responsible for passing warnings to Warnings NG. It was also necessary to create a user in Keycloak with the necessary access to run it correctly without jeopardising the system's security. After these steps, it is also necessary to add to the Jenkins environment variables the variables responsible for storing the Keycloak credentials and client, the Keycloak host and the API-Key of the vulnerability DB. These variables are passed on by Jenkins, since they are encrypted, it is not possible to print them out on screen (Jenkins does not allow this) and since they are relative to each user, it is possible to keep the credentials accessible and secure. After this last step, all that remains is to run the pipeline script and wait for the result. We then ran the tests against version 19.0.1 and obtained the results shown in the Figure 6.7.

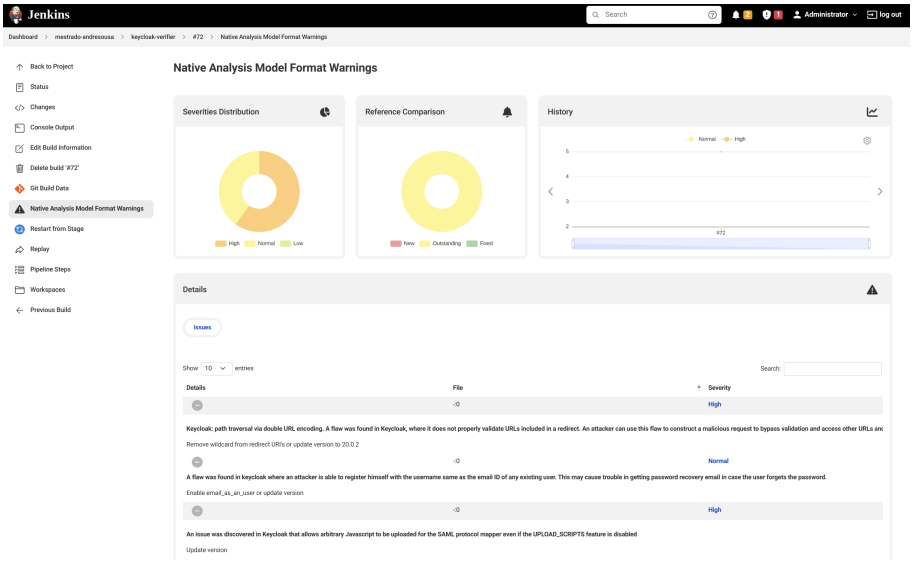


Figure 6.7: Warnings NG interface with Keycloak 19.0.1.

The test performed showed the presence of some possible vulnerabilities in the reference architecture, which would be expected given that the version in question has some associated problems. It is clear that the suggested solution produced by Keycloak Verifier for the last possible vulnerability does not specify the version to which the update should be made, however this question has now been resolved, and the secure version to which the update should be made is now presented. To make it easier for users to research, the CVE is now also provided in the first message (bold message).

There are several vulnerabilities present, but the focus will be on "CVE-2021-3754", since of those presented, it is the one with a workaround. This was resolved in version

19.0.2, however the version update may conflict with other components, in which case there is the possibility of enabling the "email_as_an_user" variable, forcing the email to be used as the username, preventing the attack referred to in the CVE, thus mitigating this vulnerability in an alternative and possibly less invasive way than the version update. Another test was carried out, this time running the same version, but locally, so that the suggested changes could be made. The result obtained was the same as the Figure 6.7, but this time without an interface, and the result can be seen in the output file. The changes suggested above were then made, enabling "email_as_an_user" and disabling "login_with_email", resulting in the configurations shown in the Figure 6.8.

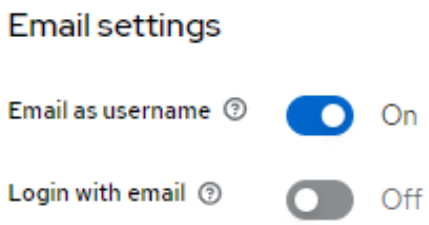


Figure 6.8: Realm settings of a good realm.

Once these changes had been made, the plugin was run again and this time the vulnerability in question was not present in the result, thus proving not only that it had been correctly detected, but also verified, ensuring that if the configuration does not allow the vulnerability, it will not be issued.

Regarding performance, the first time the plugin ran, the times observed were around one minute and forty seconds, including getting the plugin from GitHub, compiling it and everything else. In the following iterations, since Jenkins detects the changes made to the code on GitHub, there is not always the need to download and compile, so the execution times reduce to around one minute, maybe less, varying with issues such as the response times of the REST requests to DB and Keycloak, and also the existence or not of checks to be executed.

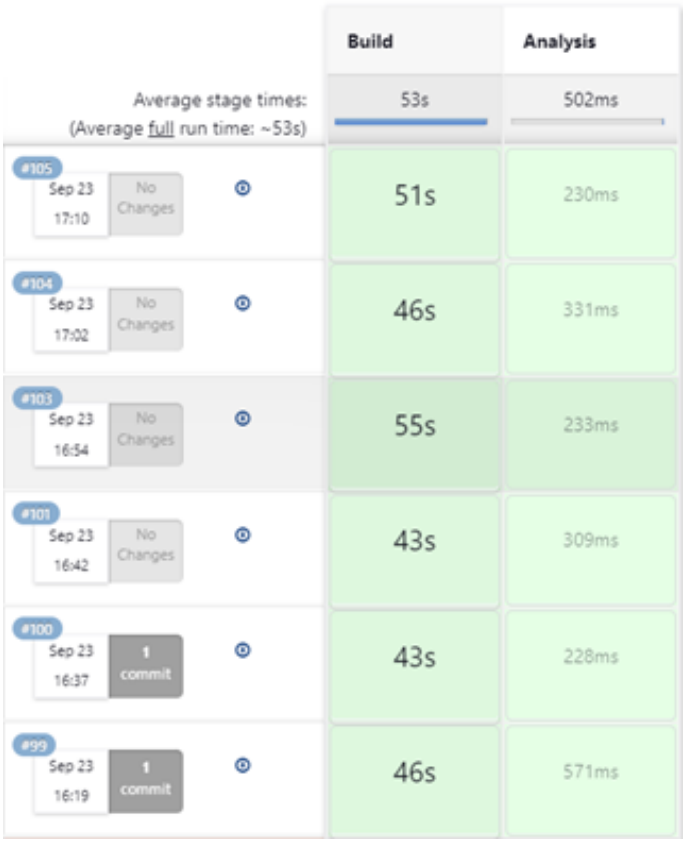


Figure 6.9: Execution times of the Jenkins' pipeline with Keycloak Verifier.

The Figure 6.9 shows an extract from the various tests carried out. For this purpose, the various versions described above were used, hence the differences between commits, it was necessary to change one or other parameter, namely the HOST to access Keycloak. It is possible to see that the execution times after the first iteration do not exceed one minute. The "build" phase is the most time-consuming, since it is necessary to obtain the code, compile it and run it. The second phase is the shortest, as it only runs Warnings NG with the output file previously produced.

In the Figure 6.10 can be seen an extract of the interface generated by Warnings NG. In the top left-hand corner can be read "Native Analysis Model Format", since the warnings generated by Keycloak Verifier have been converted into the native format of Warnings NG, according to the documentation [59]. Below 6.10 are some graphs that WNG produces, the first being a distribution of vulnerabilities by severity, which can make it possible to identify and resolve the vulnerabilities considered most threatening first. The graph in the middle makes it easier not only to detect the resolution of vulnerabilities, but also to detect the appearance of new vulnerabilities. Finally, the graph on the right shows the variation in the number of vulnerabilities present in the instance, making it possible to identify the frequency of emergence and mitigation of vulnerabilities over time.

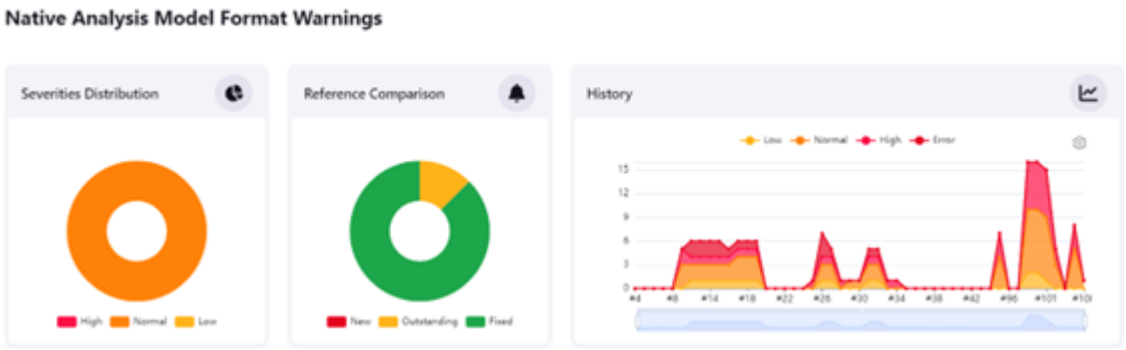


Figure 6.10: Warnings NG graphs created.

After carrying out and analysing the above tests, it is considered possible to prove the reliability and quality of Keycloak Verifier, since it was able to detect existing vulnerabilities, verify them, thus reducing the number of false positives, suggesting changes when possible, or just updating the version in other cases, resulting in an interface capable of intuitively conveying the vulnerabilities as well as the solutions. Looking at the execution times, the size of the DB required, the security of credentials and accesses, as well as the other points mentioned above, it is considered that a useful tool has been created, with a market to fit into, lightweight, free of charge and, above all, competent and with proven results. Although this tool is constantly being developed, given the emergence of new vulnerabilities and the existence of a local DB in the project, which is updated manually, it is hoped that it will easily evolve and adapt to the vulnerabilities that emerge, thus enabling it to keep Keycloak users safe, or at least aware of possible vulnerabilities and solutions, with periodic reviews. It is also considered, after analysing the experimental results, that the task has been successfully completed and the tool will be useful for guaranteeing the security of the reference architecture.

7

Conclusions

With the last chapter now over, it is time to summarise the issues covered so far and draw some conclusions from this work. The process began with an analysis and survey of the reference architecture, in order to understand the existing components and the possible vulnerabilities that might have to be dealt with. Once this analysis had been carried out, it was concluded that some vulnerabilities could currently be present, or could arise in the future. Once this data had been collated, it was necessary to come up with a solution that could help in some way, while not being too intrusive, and never forgetting that this is a business environment. It was then concluded that there were some tools in use that could prove problematic, as they had a tendency for vulnerabilities to appear. After research, it was realised that Jenkins was crucial in the development of code in the department, and a solution could be related to Jenkins. This led to the proposal to create a plugin that would carry out the necessary checks on Keycloak to guarantee its security. Since it is responsible for access, it would be a key part of protecting the architecture. In addition, other possible vulnerabilities were discovered in other components, so an analysis was made of the frequency of vulnerabilities appearing in other components of the architecture.

Taking these issues into account, the tools in possible danger were then analysed in detail, and the vulnerabilities mentioned were mitigated by updating the version of the components, and since they presented a low rate of vulnerabilities, the issue was considered resolved. Allied to this, a plugin was then developed, Keycloak Verifier, capable of analysing a Keycloak instance and, by comparing it with CVEs found up to the date of each use, detecting the presence of vulnerabilities or not. In addition to detecting vulnerabilities, for CVEs in which there is a verification to be done, it will be carried out before the alert is issued, so that the number of false alerts is reduced. The plugin then outputs a file with objects of a standard type, containing information such as the date, description, solution, etc. This data is then integrated with Warnings NG, so that a graphical output is presented, allowing users to better understand the data. Using Jenkins, it is possible to integrate the two using simple commands, thus obtaining the aforementioned interface, where it is possible to analyse issues such as description, solution, etc. Some problems have arisen, which have led to paths being taken that have led

to the current solution. Despite this, the solution found fulfils the requirements initially presented, of increasing the security of Link's reference architecture.

Although the plugin has been finalised and tested, the work is not considered to be complete, since due to external factors the interface was not exactly as expected. For this reason, an error will be reported, and it is hoped that it will be resolved, after which the necessary changes will be made so that the expected interface is obtained. The solution found is considered to be easily scalable and a valid skeleton for the creation of similar tools. In addition, its value to Keycloak users is considered to be sufficient for the implementation of the open-source solution, both for Keycloak users, developers of future similar plugins and those who wish to integrate their plugin with Warnings NG. We believe that the Keycloak Verifier will help not only Link, but also many other Keycloak users, obtaining both the tool and the verifier at no extra cost, and will be another step in the open-source community.

Having looked at these issues, it is considered that the aim of the work has been fulfilled, obtaining not only an architecture with no known vulnerabilities to date, but also a plugin capable of securing Keycloak. The solution can and should be altered and improved, and the project will not end here, but rather make small, simple changes so that the solution grows over time, since care has been taken to make the solution easily scalable.

Bibliography

[1] URL: https://www.synopsys.com/glossary/what-is-sast.html (cit. on pp. 6, 7).

[2] URL: https://www.synopsys.com/glossary/what-is-dast.html (cit. on pp. 7, 8).

[3] URL: https://www.microfocus.com/en-us/what-is/dast (cit. on p. 7).

[4] URL: https://www.veracode.com/security/interactive-application-security-testing-iast (cit. on pp. 9, 11).

[5] URL: https://www.synopsys.com/glossary/what-is-software-composition-analysis.html (cit. on p. 9).

[6] URL: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-runtime-application-self-protection-rasp/#UsecaseforRASP (cit. on p. 9).

[7] URL: https://tudip.com/blog-post/dynamic-application-security-testing/ (cit. on p. 10).

[8] URL: http://www.sonarsource.com/products/sonarqube (cit. on pp. 10, 22).

[9] URL: https://www.sonarsource.com/ (cit. on p. 10).

[10] URL: https://www.redhat.com/pt-br/topics/devops/what-cicd-pipeline (cit. on p. 11).

[11] URL: https://www.keycloak.org (cit. on p. 11).

[12] URL: https://www.keycloak.org/docs/latest/server_admin/ (cit. on p. 12).

[13] URL: https://konghq.com/products/api-gateway-platform (cit. on p. 12).

[14] URL: https://www.postman.com/ (cit. on p. 12).

[15] URL: https://www.nginx.com/learn/api-gateway/ (cit. on p. 12).

[16] URL: https://spring.io/web-applications (cit. on p. 13).

[17] URL: https://www.devmedia.com.br/exemplo/como-comecar-com-spring/73 (cit. on p. 13).

[18] URL: https://github.com/jenkinsci/warnings-ng-plugin/blob/master/doc/Documentation.md (cit. on p. 14).

[19] URL: https://www.veracode.com/products/binary-static-analysis-sast (cit. on p. 22).

[20] URL: https://www.veracode.com/why-veracode/for-security (cit. on p. 22).

[21] URL: https://checkmarx.com (cit. on p. 22).

[22] URL: https://rules.sonarsource.com/java/RSPEC-2068 (cit. on p. 23).

[23] URL: https://www.sonarsource.com/open-source-editions/ (cit. on p. 23).

[24] URL: https://loc.bitbucket.io/pulse-x.pdf (cit. on p. 23).

[25] URL: https://fbinfer.com/docs/about-Infer (cit. on p. 23).

[26] URL: https://katalon.com/resources-center/blog/ci-cd-tools (cit. on p. 24).

[27] URL: https://circleci.com/product/ (cit. on p. 24).

[28] URL: https://teamcity.com/gotime/ (cit. on p. 24).

[29] URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Keycloak (cit. on p. 27).

[30] URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-31655 (cit. on p. 30).

[31] URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-41053 (cit. on p. 30).

[32] URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-36824 (cit. on p. 30).

[33] URL: https://www.elastic.co/pt/kibana (cit. on p. 32).

[34] URL: https://grafana.com/grafana/ (cit. on p. 32).

[35] URL: https://gitee.com/jenkins-zh/warnings-ng-plugin/blob/master/doc/Documentation.md#export-your-issues-into-a-supported-format (cit. on p. 40).

[36] URL: https://github.com/jenkinsci/warnings-ng-plugin/blob/master/doc/Documentation.md (cit. on p. 40).

[37] 2019. URL: https://www.codegrip.tech/productivity/what-is-code-vulnerability/ (cit. on p. 20).

[38] 2023. URL: https://www.cncs.gov.pt/pt/sobre-nos/ (cit. on p. 1).

[39] 2023. URL: https://docs.veracode.com/r/c_about_pipeline_scan (cit. on p. 11).

[40] 2023. URL: https://github.com/insidersec/insider (cit. on p. 22).

[41] 2023. URL: https://jfrog.com/devops-native-security/ (cit. on p. 28).

[42] K. Aebersold. *Software Testing Methodologies*. 2019. URL: https://smartbear.com/learn/automated-testing/software-testing-methodologies/ (cit. on p. 5).

[43] Anchises. *Dast OU SAST*. 2019. URL: https://anchisesbr.blogspot.com/2019/05/seguranca-dast-ou-sast.html (cit. on p. 8).

[44] *Apache Kafka : Security vulnerabilities, CVEs*. 2023. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-48980/Apache-Kafka.html?page=1&order=7&trc=7&sha=63a1bba26691b3fe7ddd31f667b76c214d3ec7a9 (cit. on p. 32).

[45] C. Aschermann. *Incorrectness Logic by Example*. 2020. URL: https://hexgolems.com/2020/04/incorrectness-logic-by-example/ (cit. on p. 23).

[46] Avi. *Jenkins vs TeamCity: The Better CI Tool in 2023?* 2021. URL: https://geekflare.com/jenkins-vs-teamcity/ (cit. on p. 25).

[47] C. N. de Cibersegurança. *Quadro Nacional de Referência para Cibersegurança*. 2022. URL: https://www.cncs.gov.pt/docs/cncs-qnrcs-2019.pdf (cit. on p. 1).

[48] Clearlinux. *Clearlinux/CVE-check-tool: Original automated CVE checking tool*. URL: https://github.com/clearlinux/cve-check-tool (cit. on p. 28).

[49] *Como usar o OpenVAS para avaliação de vulnerabilidades*. 2019. URL: https://www.welivesecurity.com/br/2019/07/24/como-usar-o-openvas-para-avaliacao-de-vulnerabilidades/ (cit. on p. 26).

[50] *Create a Plugin*. 2023. URL: https://www.jenkins.io/doc/developer/tutorial/create/ (cit. on p. 35).

[51] *CVE Search API*. URL: https://docs.cvesearch.com/ (cit. on p. 41).

[52] DevopsCurry. *Jenkins is getting Old, so what are the alternatives in 2021 ?* 2020. URL: https://medium.com/devopscurry/jenkins-is-getting-old-so-what-are-the-alternatives-in-2021-fd6ce6707465 (cit. on p. 25).

[53] Greenbone. *Greenbone/openvas-scanner: This repository contains the scanner component for Greenbone Community Edition.* URL: https://github.com/greenbone/openvas-scanner (cit. on p. 26).

[54] *GREV*. 2023. URL: https://hanadigital.github.io/grev/hanadigital.github.io/grev/ (cit. on p. 29).

[55] C. Griffiths. *The Latest 2022 Cyber Crime Statistics (updated December 2022) | AAG IT Support*. 2023. URL: https://aag-it.com/the-latest-cyber-crime-statistics/ (cit. on p. 1).

[56] T. Hamilton. *What is WHITE Box Testing? Techniques, Example, Types Tools*. 2019. URL: https://www.guru99.com/white-box-testing.html (cit. on p. 6).

[57] G. Inc. *Top Postman API Platform Likes Dislikes 2023 | Gartner Peer Insights*. URL: https://www.gartner.com/reviews/market/full-life-cycle-api-management/vendor/postman/product/postman-api-platform/likes-dislikes (cit. on p. 12).

[58] G. Jeanmart. *[keycloak-user] Username enumeration*. 2017. URL: https://lists.jboss.org/pipermail/keycloak-user/2017-October/012064.html (cit. on p. 18).

[59] Jenkinsci. *Jenkinsci/analysis-model: A library to read static analysis reports into a java object model*. URL: https://github.com/jenkinsci/analysis-model/ (cit. on p. 50).

[60] J. P. M. Jr. *What is runtime application self-protection (rasp)?* 2019. URL: https://techbeacon.com/security/what-runtime-application-self-protection-rasp (cit. on p. 9).

[61] E. Katz. *Top 10 Static Application Security Testing (SAST) Tools in 2021*. 2021. URL: https://spectralops.io/blog/top-10-static-application-security-testing-sast-tools-in-2021/ (cit. on p. 22).

[62] *Kibana 7.17.9 and 8.6.2 Security Update*. URL: https://discuss.elastic.co/t/kibana-7-17-9-and-8-6-2-security-update/325782 (cit. on p. 29).

[63] *Kibana 8.7.1 Security Updates*. 2023. URL: https://discuss.elastic.co/t/kibana-8-7-1-security-updates/332330 (cit. on p. 29).

[64] J. Knutsen. *Privilege escalation vulnerability on Token Exchange feature*. 2022. URL: https://github.com/keycloak/keycloak/security/advisories/GHSA-75p6-52g3-rqc8 (cit. on p. 19).

[65] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[66] *Maven – Introduction to Maven Plugin Development*. 2023. URL: https://maven.apache.org/guides/introduction/introduction-to-plugins.html (cit. on p. 35).

[67] P. Mishra. *21 Of The Best Jenkins Alternatives For Developers*. 2020. URL: https://www.lambdatest.com/blog/best-jenkins-alternatives/ (cit. on pp. 24, 25).

[68] J. Mulliken. *2115392 – (CVE-2022-2668) CVE-2022-2668 keycloak: Uploading of SAML javascript protocol mapper scripts through the admin console*. 2022. URL: https://bugzilla.redhat.com/show_bug.cgi?id=2115392 (cit. on p. 20).

[69] *Nessus Documentation | Tenable™*. 2023. URL: https://docs.tenable.com/Nessus.htm (cit. on pp. 25, 26).

[70] *NPM Audit + Jenkins Warnings Next Generation (Custom Groovy Parser)*. 2023. URL: https://uko.codes/npm-audit-jenkins-warnings-next-generation-custom-groovy-parser (cit. on p. 40).

[71] *NVD - CVE-2022-38779*. 2022. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-38779 (cit. on p. 29).

[72] *NVD - CVE-2023-31414*. 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2023-31414 (cit. on p. 29).

[73] H. dr nyt. *GREV*. URL: https://hanadigital.github.io/grev/?user=keycloak&repo=keycloak (cit. on p. 19).

[74] *OpenVAS - Open Vulnerability Assessment Scanner*. 2023. URL: https://openvas.org/ (cit. on pp. 26, 33).

[75] OWASP. *A01 Broken Access Control - OWASP Top 10:2021*. 2021. URL: https://owasp.org/Top10/A01_2021-Broken_Access_Control/ (cit. on p. 20).

[76] OWASP. *OWASP Top Ten*. 2021. URL: https://owasp.org/www-project-top-ten/ (cit. on p. 20).

[77] *P2PInfect: The Rusty Peer-to-Peer Self-Replicating Worm*. 2023. URL: https://unit42.paloaltonetworks.com/peer-to-peer-worm-p2pinfect/ (cit. on p. 30).

[78] Pantsel. *More than just another kong gui*. URL: https://pantsel.github.io/konga/ (cit. on p. 12).

[79] *Pipeline Maven Integration*. 2023. URL: https://plugins.jenkins.io/pipeline-maven (cit. on p. 40).

[80] *Pipeline Maven Integration Plugin*. URL: https://www.jenkins.io/doc/pipeline/steps/pipeline-maven/ (cit. on p. 40).

[81] S. Q. *Keycloak*. URL: https://www.surecloud.com/resources/blog/pentesting-keycloak-part-1 (cit. on p. 19).

[82] M. Rehak. *2013577 – (CVE-2021-20323) CVE-2021-20323 keycloak-services: POST based reflected Cross Site Scripting vulnerability*. 2021. URL: https://bugzilla.redhat.com/show_bug.cgi?id=2013577 (cit. on p. 19).

[83] K. S. *Cross Site Scripting (XSS) | OWASP*. 2020. URL: https://owasp.org/www-community/attacks/xss/ (cit. on p. 19).

[84] *TechDoc Portal*. URL: https://docs.greenbone.net/ (cit. on p. 26).

[85] *Tenable Vulnerability Management (Formerly Tenable.io)*. 2023. URL: https://www.tenable.com/products/tenable-io (cit. on p. 26).

[86] *Vuls: VULnerability Scanner*. 2023. URL: https://github.com/future-architect/vuls (cit. on p. 33).

[87] *Warnings Next Generation*. 2023. URL: https://plugins.jenkins.io/warnings-ng (cit. on p. 13).

[88] *Warnings Next Generation Plugin*. 2023. URL: https://www.jenkins.io/doc/pipeline/steps/warnings-ng/ (cit. on pp. 13, 14).