Project Number 101017258

D5.5 Security Analysis of EDDIs

Version 1.0
5 July 2023
Final

Public Distribution

FORTH

Project Partners: **Aero41, ATB, AVL, Bonn-Rhein-Sieg University, Cyprus Civil Defence, Domaine Kox, FORTH, Fraunhofer IESE, KIOS, KUKA Assembly & Test, Locomotec, Luxsense, The Open Group, Technology Transfer Systems, University of Hull, University of Luxembourg, University of York**

PROJECT PARTNER CONTACT INFORMATION

Aero41 Frédéric Hemmeler Chemin de Mornex 3 1003 Lausanne Switzerland E-mail: frederic.hemmeler@aero41.ch	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany E-mail: scholze@atb-bremen.de
AVL Martin Weinzerl Hans-List-Platz 1 8020 Graz Austria E-mail: martin.weinzerl@avl.com	**Bonn-Rhein-Sieg University** Nico Hochgeschwender Grantham-Allee 20 53757 Sankt Augustin Germany E-mail: nico.hochgeschwender@h-brs.de
Cyprus Civil Defence Eftychia Stokkou Cyprus Ministry of Interior 1453 Lefkosia Cyprus E-mail: estokkou@cd.moi.gov.cy	**Domaine Kox** Corinne Kox 6 Rue des Prés 5561 Remich Luxembourg E-mail: corinne@domainekox.lu
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**Fraunhofer IESE** Daniel Schneider Fraunhofer-Platz 1 67663 Kaiserslautern Germany E-mail: daniel.schneider@iese.fraunhofer.de
KIOS Maria Michael 1 Panepistimiou Avenue 2109 Aglatzia, Nicosia Cyprus E-mail: mmichael@ucy.ac.cy	**KUKA Assembly & Test** Michael Laackmann Uhthoffstrasse 1 28757 Bremen Germany E-mail: michael.laackmann@kuka.com
Locomotec Sebastian Blumenthal Bergiusstrasse 15 86199 Augsburg Germany E-mail: blumenthal@locomotec.com	**Luxsense** Gilles Rock 85-87 Parc d'Activités 8303 Luxembourg Luxembourg E-mail: gilles.rock@luxsense.lu
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium E-mail: s.hansen@opengroup.org	**Technology Transfer Systems** Paolo Pedrazzoli Via Francesco d'Ovidio, 3 20131 Milano Italy E-mail: pedrazzoli@ttsnetwork.com
University of Hull Yiannis Papadopoulos Cottingham Road Hull HU6 7TQ United Kingdom E-mail: y.i.papadopoulos@hull.ac.uk	**University of Luxembourg** Miguel Olivares Mendez 2 Avenue de l'Universite 4365 Esch-sur-Alzette Luxembourg E-mail: miguel.olivaresmendez@uni.lu
University of York Simos Gerasimou & Nicholas Matragkas Deramore Lane York YO10 5GH United Kingdom E-mail: simos.gerasimou@york.ac.uk nicholas.matragkas@york.ac.uk	

DOCUMENT CONTROL

Version	Status	Date
0.1	Initial draft with outline and first content	13 May 2023
0.2	First draft	12 June 2023
0.3	Ready for internal review	28 June 2023
0.9	Updated version from internal reviews	4 July 2023
1.0	Final QA version	5 July 2023

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

This deliverable describes the techniques and tools that are adopted towards securing the EDDIS created in the context of the SESAME project. The executable nature of the EDDIs makes them extra vulnerable forcing the adoption and development of dedicated tools to ensure that no additional attack surfaces are created and no additional attack types are utilized.

The document presents the state-of-the-art techniques and tools towards ensuring that executable files stay secure. It focuses on static code analysis and hash verification techniques, describing the most popular opensource tools and presenting custom solutions, tailored to the project use cases.

The tools that are mentioned in this document are either opensource, with their code publicly available in free repositories, or custom, which are going to become available at the project's repository. A short description of each follows:

- **CheckStyle**: a static analysis tool for Java code that enforces coding conventions, detects potential bugs.
- **Checkmarx**: a comprehensive application security testing platform that identifies and remediates software vulnerabilities throughout the development lifecycle.
- **Qodana:** an intelligent code quality and security analysis platform that provides automated code inspections and identifies potential issues.
- **SonarQube:** an open-source platform for continuous code quality inspection and static analysis, enabling developers to detect and resolve code issues, ensure coding standards, and enhance overall code quality.
- **Hash verification Python script:** a script that calculates hash values for a specified file using multiple hash algorithms and compares them with pre-defined hash values for verification, providing information on whether the hash values match or not.

LIST OF ABBREVIATIONS

CAPEC	Common Attack Pattern Enumeration and Classification
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DoS	Denial-of-Service
EDDI	Executable Digital Dependability Identity
NVD	National Vulnerability Database
IDE	Integrated Development Environment

1. INTRODUCTION

1.1 OVERVIEW

Ensuring the security of an executable is critical since it can be an entry point for an attacker that aims to compromise the security of the whole of our system. A malicious executable could lead to malevolent actions such as malware propagation, unauthorized system access, or compromising user privacy. A compromised system cannot protect valuable data and assets from unauthorized access or theft and could lead to financial loss, reputational damage, and legal consequences. A well-protected system contributes to business continuity by minimizing disruptions, crashes, and performance issues, and strengthens compliance with industry regulations and standards. Non-compliance can result in penalties and legal consequences. Finally, demonstrating a commitment to security, you build trust with users, customers, and partners, enhancing your reputation.

Executables differ from other assets in terms of security due to their active code execution, privileged access requirements, complexity, distribution, and attractiveness to attackers. Executables contain code that is actively executed on a system, interacting with the underlying operating system and network, introducing a higher level of risk. They often require elevated privileges and access to sensitive resources, making their security crucial to prevent unauthorized activities. Moreover, they are depending on external libraries and frameworks, increasing the attack surface and the number of potential vulnerabilities. Additionally, executables are often distributed and executed on multiple systems, making themselves vulnerable to tampering. Due to all of the above reasons, executables are very popular targets for attackers that aim to gain control over systems, compromise data, or propagate malware. Due to their potential impact and wide distribution, executables are frequently targeted by attackers. Protecting executables involves implementing specialized security measures such as code review, digital signatures, encryption, and testing to mitigate these unique risks.

1.2 DELIVERABLE STRUCTURE

The rest of the deliverable is structured as follows. Section 2 cover techniques, tools, and best practices related to guaranteeing that executable files remain secure. The focus of the section is on the concepts of static code analysis and hash verification, both for ensuring security in executable files. Regarding static security analysis, the concept is explained and individual tools are described. As far as the hash verification is concerned, once again the concept is explained, mentioning its role in verifying file integrity, and how it can be used to detect tampering or unauthorized changes.

Section 3 delves deeper into the tools that have been adopted until now for making the SESAME EDDIs secure. A set of chosen static analysis tools are described in details with screenshots that show their functionalities are created reports. Moreover, listings with Python scripts are presented, with our custom development of a hash verification tool.

Finally, we present our concluding remarks in section 4.

2. MAKING EXECUTABLE FILES SECURE

As it is already discussed, we need to focus on the security of executables due to their active code execution, privileged access requirements, complexity, distribution, and the fact that attackers seem to prefer them as targets. A number of different techniques that can be utilized for securing executables are mentioned below:

Code review is a critical process for ensuring the security of an executable. It involves a thorough inspection of the source code to identify security vulnerabilities, coding errors, and potential risks. Common security issues that can be detected include injection attacks, cross-site scripting, and insecure authentication mechanisms. Additionally, code review helps improve overall code quality, identify bugs, encourages knowledge sharing, and ensures adherence to coding standards. Code review can improve code quality, enhance security practices, and reduce the likelihood of security breaches.

Digital signature is a cryptographic mechanism used to verify the authenticity, integrity, and non-repudiation of digital content, such as executables. Using asymmetric cryptography, a signer with their private key encrypts a hash value of the content, creating a digital signature. The produced digital signature, along with the signer's public key, are then attached to the content. Recipients can verify the signature using the signer's public key to authentication, integrity, and non-repudiation, along with the assurance that the content comes from a trusted source.

Antivirus software and malware scanning constitute another way to ensure the security of executables. Antivirus programs use different processes to identify and mitigate known and unknown threats, such as signature-based detection, heuristic analysis, behaviour monitoring, and sandboxing. They compare file signatures, analyse code behaviour, monitor system activity, and execute executables in isolated environments to detect and block malware. Malware scanning takes advantage of some of the aforementioned techniques to identify and remove malicious code. Antivirus software and malware scanning are able to provide real-time protection and help prevent the execution of malware.

Hash verification is a mechanism for checking the integrity of data using cryptographic hash functions. A hash function takes input data and produces a fixed-size hash value or checksum. The generated hash value is unique to the input data. By comparing the computed hash value of the received data with the expected hash value, the integrity of the data can be verified. In that way it is ensured that target data has not been altered or corrupted during the time window between the generation of the two hash values, the expected and the new one.

Security testing is a term that includes a number of techniques that can be used for ensuring executable security, such as vulnerability identification, risk mitigation, compliance with standards, penetration testing, and secure lifecycle development. Security testing simulates real-world attacks to evaluate the effectiveness of security controls and identify areas for improvement.

Our work in SESAME task T5.4 focuses on two of the techniques presented above, code review and hash verification.

2.1 STATIC CODE ANALYSIS

It could be said that the history of the static code analysis tools starts with Lint, a tool created by Stephen Johnson at the Bell Laboratories in the 1970s. The goal of Lint was to scan C source programs with no compilation errors and identify unnoticed bugs. It scanned the source code for matches without running the actual program. Tools with that functionality are called static checkers. The dynamic way of checking for errors includes running the program and compare the actual with the expected behaviour. The facts that people tend to make the same mistakes over and over, and most errors belong to known categories, makes it possible for static checkers to produce meaningful results.

There are two kinds of static code checkers: those that work on the source code directly and those that work on the compiled bytecode, each with its own advantages and disadvantages. The checkers that work on the source code directly search for matches at the exact code that the programmer wrote. On the other hand, when a program is compiled, the actual code is optimized at some degree and the produced bytecode might not match the source. However, working on bytecode is much faster, a critical factor for projects with a large volume of code lines. The way most code checkers work includes the build of an abstract representation of the target program (model), and a data-flow analysis to figure out the possible values that variables might have [1].

The static analysis tools that are mentioned by Bardas [2] are called source code analyzers, which should spot code weaknesses, reporting their location and severity. The weakness class should match a Common Weakness Enumeration (CWE) entry. Additional information that could be reported include conditions that are necessary for the weakness to be revealed, related data or control flow, fix suggestions, and a false positive index.

Bardas mentions a number of static code analysis' advantages and disadvantages compared with the dynamic analysis (testing):

- Static analysis can be performed only on specific modules or on code that is still unfinished, even during the development process. Testing needs modules that can be executed, test cases or input data, possibly supporting drivers, and auxiliary components. Testing, due to its nature, can be conducted when the target program is in a mature phase.

- Static analyzers consider code independently of any particular execution. Being able to enumerate all possible interactions between the different modules and components, they may reveal rare occurrences or hidden back doors. In case of testing, establishing of initial conditions or artificial constrain to the system may needed to produce a desired interaction.

- Static analyzers have to deal with the limitations of their reasoning sophistication. A good analyzer should be able to identify a large range of weaknesses. For testing, the same problem is addressed with the development of the corresponding tests that can exercise a particular property or module.

- In case of the discovery of new attacks or failures, a static analyzer just needs to update its vulnerability database. An analyzer with outdated database will miss the new weakness in the analyzed code. Once the database is updated, the analyzer can start checking for matches to all target code with no limitations. On the other hand, new tests need to be created for identification of new vulnerabilities. Although updating of the vulnerability database is easier, there will be always complex vulnerabilities that could not be detects by static analyzers, such as the lack of auditing or encryption.

According to Gomes et al. [3], static code analysis is divided into i) Manual Review, and ii) Usage of automated tools. The former consists of a series of phases including the actual implementing of the code, the self-review, the walkthrough, the peer review, and the inspection and audit. During self-review, the programmer tries to spot code errors on its own. The walkthrough is the phase during which the programmer presents their code to an audience. During peer review, a colleague of the programmer reviews the code. Finally, a party of evaluators review the code during the inspecting and audit phase. This way of code analysis is time-consuming, while it requires the reviewers to be aware of the type of error they are going to try to find.

On the other hand, the usage of dedicated tools for the aforementioned task, can produce results much faster, without the need for high level of expertise from the tool operators. However, expertise is needed from the authors of the rules that are going to be used by the automated tools for the security problem detection. A tool needs a set of rules able to detect a large range of security problems to be considered effective.

Gomes et al. presents the advantages and disadvantages of the static code analysis compared with the dynamic way of detecting code security problems. Results of dynamic analysis cannot be generalized due to the set of inputs that were used for their production. Used input most probably does not represent all possible program executions. On the contrary, the results of a static analysis are able to describe in an accurate way the target program’s behaviour, since they are input- and execution environment- agnostic. A very useful advantage of the static code analysis is that it can be conducted in the very early stages of the code production, forcing it to be reliable and lees prone to errors.

Authors in [4] provide a list of types of problems that static code analyzers can detect: syntactic problems, unreachable source code, undeclared variables, non-initialized variables, non-used functions and procedures, variables used before initialization, nonuse of values from functions, wrong use of pointers.

A taxonomy of static code analysis tools is presented in [5]. The authors tried to classify the available tools using a taxonomy tree. The categories of the tree where created based on the input that the tools accept, the number of their releases per year, the supported languages, the searching technologies, the set of supported rules, the tool configurability, the rule extensibility, the tool availability, the user experience factor, and finally, the way the search results are presented. The created taxonomy tree can be seen in Figure 1.

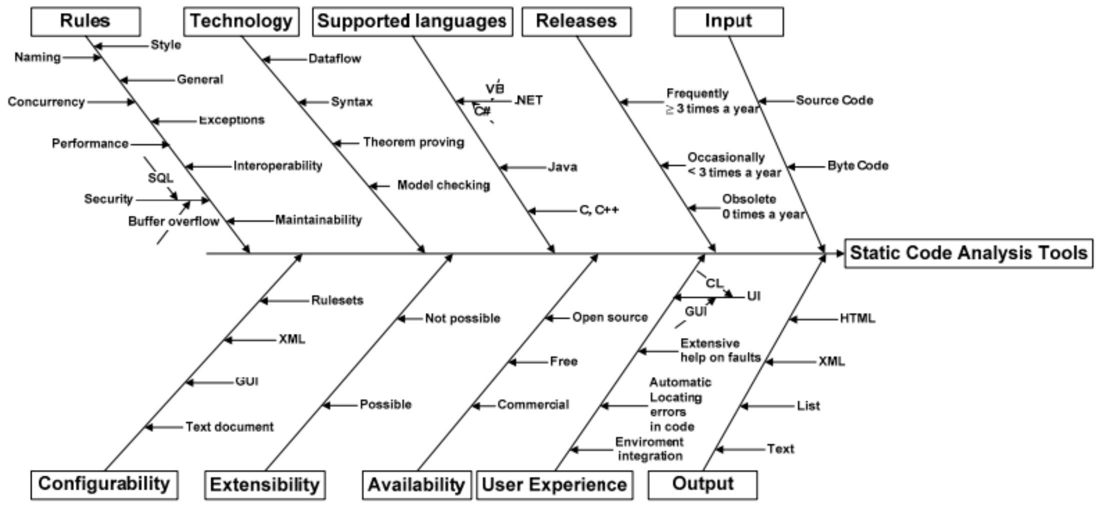


Figure 1: Taxonomy tree of static code analysis tools [5]

Static code analysis tools seem a very promising solution that can support automatic detection and being scalable at the same time. However, how effective are they? This is the question that authors in [6] tried to answer. They present an empirical evaluation of the ability of static code analysis tools to detect security vulnerabilities. The Juliet benchmarking test suite was used for the evaluation of three commercial static code analysis tools. Juliet test suite consists of many sets of synthetically generated test cases; each set covers only one kind of flaw documented by the Common Weakness Enumeration (CWE). The evaluation methodology included 6 steps and according to the results, 27% of C/C++ vulnerabilities and 11% of Java vulnerabilities were missed by all three tools. Some vulnerabilities were detected by only one or combination of two tools; 41% of C/C++ and 21% of Java vulnerabilities were detected by all three tools. This conclusion suggests that static code analysis could leave a number of vulnerabilities undiscovered.

A comparison among three more static code analysis tools was conducted in [7]. Fortify SCA, Splint, and Frama-C were compared by analyzing their performance when checking a demonstration code. The demonstration code, was originally error free but later some errors were introduced, with annotations. The introduced errors included buffer overflow, memory handling, dereference errors, and control flow errors. According to the results, Frama-C was able to discover all the errors, however, giving at the same time many false positives. Splint missed some of the errors but was easier to adopt. Finally, Fortify SCA missed at least one error, but produced no false positives. Additional conclusions are that static code analyzers can be useful by removing errors but their output needs be further analyzed and be understood by the tool users to avoid unwanted consequences. tools based on annotations have good potential but demand more of their users.

Six different static analysis tools are described and compared in [8]. The selection of the tools has been made based on the familiarity of the authors with them and their popularity and wide adoption. The selected tools are Better Code Hub, Checkstyle, Coverity scan, FindBugs, PMD, and SonarQube. According to their findings, false positives is the main issue of all participating tools. That affects the overall precision that seems to range from 18% to 57%. Noticeable exemption is Checkstyle with precision 86%. However, the majority of the its rules are related with documentation and not functional parts of the code. Moreover, it seems that the detection agreement among the different tools is low. The highest percentage of agreement between two tools is just 9.378% (FindBugs - PMD).

Better Code Hub was a well-known static analysis tool for code quality assessment. However, it is not available any more. The analysis was done through the website's API, which used to analyze the repository from GitHub and other popular version control systems, based on ten guidelines. These guidelines were derived from industry best practices and used to cover various aspects of code quality, including maintainability, testability, simplicity, and modularity. The 10 Guidelines for Better Code, upon which Better Code Hub was based, were: Write Short Units of Code, Write Simple Units of Code, Write Code Once, Keep Unit Interfaces Small, Separate Concerns in Modules, Couple Architecture Components Loosely, Keep Architecture Components Balanced, Keep Your Codebase Small, Automate Tests, and Write Clean Code. Each guideline used to correspond to a Better Code Hub rule. The rules were grouped in 3 types: RefactoringFileCandidateWithLocationList; RefactoringFileCandidate; and RefactoringFileCandidateWithCategory, and 3 severity levels: Medium; High; Very High. The compliance with the above guidelines could be measured on a scale from 1–10 based on the results. The overall analysis of the target code was done against heuristics and popular coding conventions, providing a view of the health of the code macroscopically.

Checkstyle is another popular static analyzer that evaluates Java code quality. Google Java Style and Sun Java Style are two different configurations for code assessment along with customized configuration files. It can be integrated with Ant or be used as a command line tool. Additionally, Checkstyle can be integrated into popular integrated development environments (IDEs) like Eclipse, IntelliJ IDEA, and others, providing real-time feedback during development. The error detection is based on 173 rules grouped in 14 types: Annotations, Block Checks, Class Design, Coding, Headers, Imports, Javac Comments, Metrics, Miscellaneous, Modifiers, Naming Conventions, Regexp, Size Violations, and Whitespace. Moreover, there is a categorization based on the severity levels: Error, Ignore, Info, and rule.

Coverity scan is an open-source static analysis tool, developed by Synopsys. A public API is used for submission of code builds to a server for assessment. The source code can be written in various programming languages, including C, C++, C#, Java, and JavaScript. The tool detects defects and vulnerabilities that are grouped by categories such as: resource leaks, dereferences of NULL pointers, incorrect usage of APIs, use of uninitialized data, memory corruptions, buffer overruns, control flow issues, error handling issues, incorrect expressions, concurrency issues, insecure data handling, unsafe use of signed values, and use of resources that have been freed. The analysis that is conducted includes the examination of all the possible paths the program may take. The classification of the rules is done in three levels: Low, Medium, and High.

FindBugs is another static analyzer for Java bytecode. A GUI is available for conducting the analysis of the code, based on bug patterns. These patterns are the result of the followings: difficult language features, misunderstood API features, misunderstood invariants when code is modified during maintenance, and garden variety mistakes. The bug patterns are classified under 9 different categories: bad practice, correctness, experimental, internationalization, malicious code vulnerability, multithreaded correctness, performance, security, and dodgy code. They are additionally ranked from 1 to 20, in four groups; the scariest, scary, troubling, and concern group.

PMD is also a static analysis tool for mainly Java, JavaScript, and Salesforce Apex. It uses a set of rules for code quality assessment according to objectives such as unused variables, empty catch blocks, unnecessary object creation, and more. The utilized rules are classified under 8 categories: best practices, code style, design, documentation, error prone, multi-threading, performance, and security. The violation of the rules is ranked from 1 to 5 with 1 being the most severe and 5 being the least. There is also the alternative of custom-made rules.

SonarQube is one of the most well-known static analysis tools for quality assessment. There are two usage options; as a service by the sonarcloud.io platform or downloaded and executed locally. A centralized dashboard is provided that allows for measuring, analyzing and improving code quality. SonarQube reports Bugs that are reliability problems, Code smells that are related to maintainability, and Vulnerabilities that are security related problems. 413 rules are utilized for the code assessment and are categorized based on their severity into Blocker, Critical, Major, Minor, and Info.

2.2 HASH VERIFICATION

“A cryptographic hash function is a one-way function that converts input data to an arbitrary length and produces an output with a fixed length [9][10][11]”. The output is called "hash value" (Figure 2).

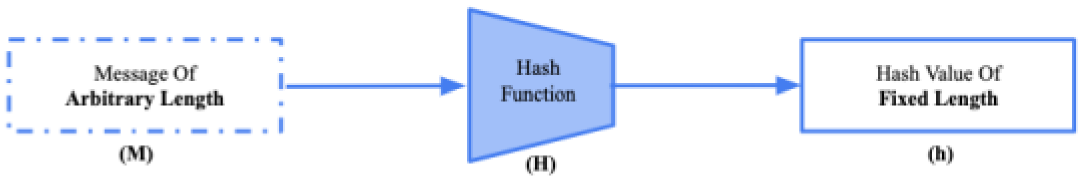


Figure 2: The basic form of a hash function [9]

The five properties that a hash needs to fulfill include:

1. The input string can be of various sizes; however, the output has a fixed length.
2. Output must be efficiently computed for any given data.
3. Be deterministic, meaning same input produces same output.
4. Data can be generated and returned from a hash value.
5. Small changes in the input data greatly affect the output hash (no correlation between old and new hashes after any change).

[9] used hash algorithms in verifying the integrity and authenticity of certificate information. Some of the conclusions of the paper are that hash functions can be used to

verify the integrity and authenticity of certificate information, index data in a hash table, and securely authenticate users without storing passwords locally.

Authors in [12] focus on the difficulty of applications that need hash to be able to localize image tampering. In such a case, hash should be small but, at the same time, must incorporate large amount of information about the original image. The paper presents an image hashing method that addresses this difficulty, managing to detect and localize tampering using a small signature (< 1kB). An image hashing method consists of hash generation and verification. Hash generation includes a set of features, is extracted from the image and a function that maps them to a bit sequence. During the verification, a query image is used, a hash is created and then compared with the original one. The proposed method, according to the authors, combines the advantages of an exhaustive search based hashing and robust representation-based hashing methods.

A software verification primitive is introduced in [13]. It is called Oblivious Hashing and allows implicit computation of a hash value based on the actual execution of the code. The proposed primitive tries to tackle disadvantages of mainstream techniques for software integrity verification, such as code checksum. To mention one, code checksum is able to verify only the static shape of the code, doing nothing about run-time attacks. According to their evaluation, Oblivious Hashing produces much lower overhead, since it hashes only critical expressions and not intermediate computations.

There a plethora of security services that take advantage of cryptographic hash functions, such as achieving integrity and authentication, are presented in [14]. Verification of the integrity and authenticity of information is a fundamental for computer systems and networks. Communicating over an insecure channel needs a validation of the authenticity of the participants and the unmodified nature of the information shared. Hash functions are also used in the digital signatures' context. Signing the hash of a message, rather than the entire message, computational overhead is reduced significantly. Moreover, Hash functions find application in user authentication. Passwords can be stored as hash values. Every time a user logs in a hash is calculated based on their password and compared with the stored one. In that way no one can access the passwords, nor even system administrators. Another application of Hash functions is digital time stamping, following the same logic. In the context of session keys, hash functions can generate a sequence of keys used to protect successive communication sessions. Starting from a master key, the hash function can be repeatedly applied to provide session keys. Some more applications of Hash functions include constructing block ciphers, index data in hash tables, perform fingerprinting, detect duplicate data or uniquely identify files, serve as checksums to detect accidental data corruption, and even generate random numbers.

3. SECURITY ANALYSIS OF SESAME EDDIs

The Executable Digital Dependability Identity (EDDI), an extension of the DDI concept, is a model-based artefact that contains all the required dependability information about a given system or component — such as safety and security hazards, their potential causes, effects, and possible corrective actions, as well as safety argumentation and information about the system architecture itself (D4.2 and D5.2). They should also support any relevant dependability activities, whether that be safety analyses, allocation of requirements, or synthesis of safety argumentation. Moreover, EDDIs are fully executable at runtime, capable of communicating and adapting to changing circumstances to help ensure continued safe and security operation.

EDDIs being executables should be treated differently in terms of security. As it was mentioned in the introduction of this document, the active code execution, privileged access requirements, complexity, distribution, and attractiveness to attackers of executable files create the need for hardening techniques that can guarantee that no new attack surfaces are offered and no newly introduced attack attempts can be conducted.

3.1 TOOLS USAGE AND DEVELOPMENT

The work conducted in the context of T5.4 focuses mainly on two of the aforementioned techniques for ensuring secure executables (EDDIs in our case): static code analysis and hash verification. Static code analysis allows for the source code examination towards identification of potential vulnerabilities, bugs, and security loopholes. Deficiencies in the structure, syntax and logic of the produced code can lead to weaknesses and attacks from adversaries.

Additionally, we utilize hash verification, the process that ensures the integrity and authenticity of executables calculating and comparing cryptographic hash values. The goal of using this technique is to detect any modifications or tampering in the executable files, providing an added layer of security.

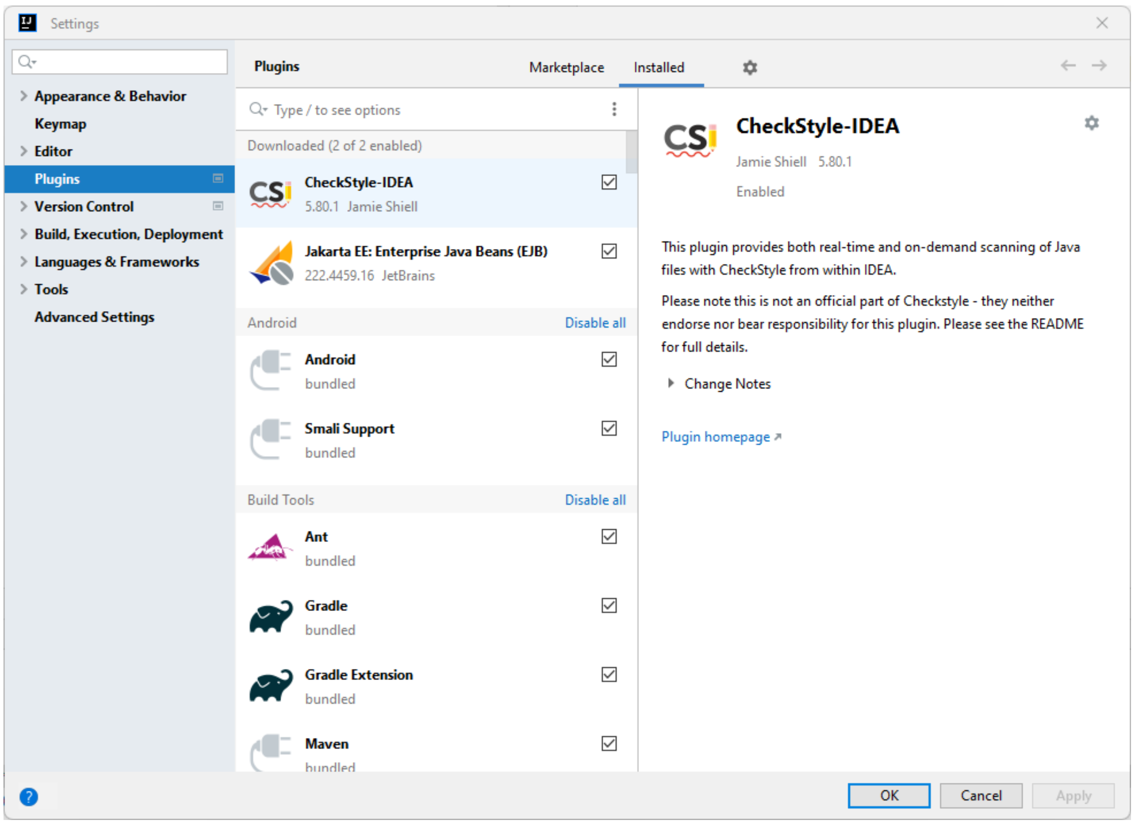
We have already mentioned in this document a number of different techniques for ensuring that executable files do not impose an additional security burden on the personnel responsible for the system's security. The goal of focusing on specific techniques is to acquire targeted and in-depth security knowledge on them towards the provision of effective protection. We are aware though that it's still crucial to have a holistic understanding of the whole set of available techniques, and incorporate them into the code development workflows for a comprehensive code verification. Most of these techniques complement each other to improve code quality and eliminate chances for bugs and vulnerabilities. In any case, combination of techniques should be based on the specific security requirements of the target system, taking under consideration costs, benefits, and trade-offs.

3.1.1 Static code analysis

There is a big boom in the static code analysis field in recent years, with production of various new tools, which focus on different programming languages, development environments, and specific requirements. The fact that there are numerous static code analysis tools available today, with a range of different features and capabilities, drove us to incorporate not only one but a set of those in the proposed technique for securing SESAME EDDIs. Testing more than one static code analyzers allowed us to evaluate and compare their features, accuracy, and performance, and choose the most suitable ones for each particular case. The tools that are presented here are CheckStyle, IntelliJ’s native analyzer (Checkmarx), Qodana, and SonarQube.

3.1.1.1 CheckStyle

CheckStyle was already described in 2.1 as a Java dedicated code quality analyser that focuses mostly on documentation and not on functional parts of the code. One of its strengths though is the fact that can be integrated into Integrated Development Environments (IDEs), providing feedback even during the development phase of the code. Since the IDE that was used by FORTH during the development of code for different tools used in the SESAME security methodology was IntelliJ IDEA, we tried to integrate CheckStyle with it and get feedback. Towards that goal, the CheckStyle-IDEA plugin should be installed in IntelliJ. Figure 3 depicts the list of installed and enabled plugins in the IntelliJ instance, in FORTH's premises.



Downloaded (2 of 2 enabled)

CheckStyle-IDEA ✓
5.80.1 Jamie Shiell

Jakarta EE: Enterprise Java Beans (EJB) ✓
222.4459.16 JetBrains

Android

Android ✓
bundled

Smali Support ✓
bundled

Build Tools

Ant ✓
bundled

Gradle ✓
bundled

Gradle Extension ✓
bundled

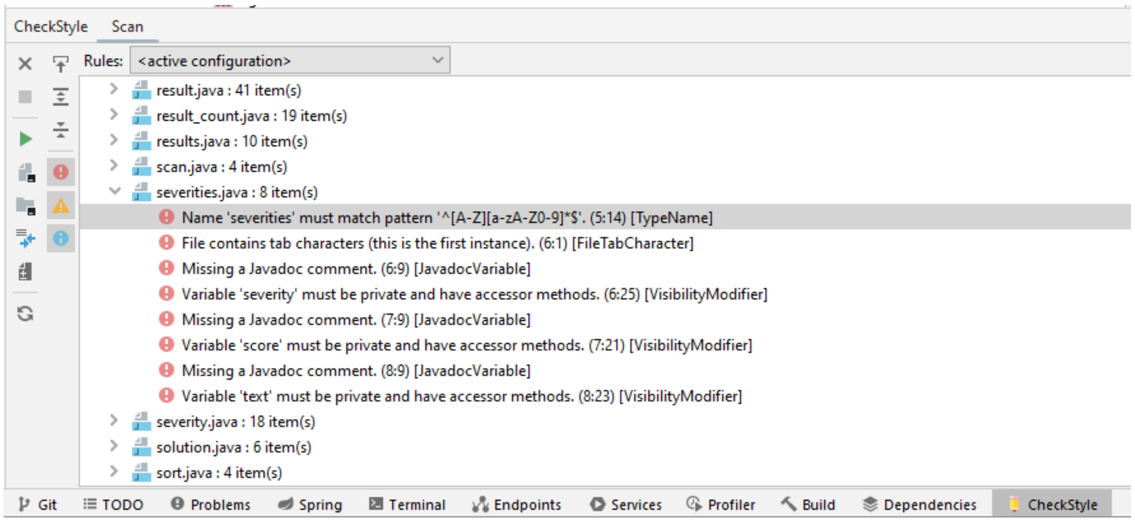
Maven ✓
bundled

CheckStyle-IDEA
Jamie Shiell 5.80.1
Enabled

This plugin provides both real-time and on-demand scanning of Java files with CheckStyle from within IDEA.

Please note this is not an official part of Checkstyle - they neither endorse nor bear responsibility for this plugin. Please see the README for full details.

Figure 3: Installing CheckStyle-IDEA to IntelliJ



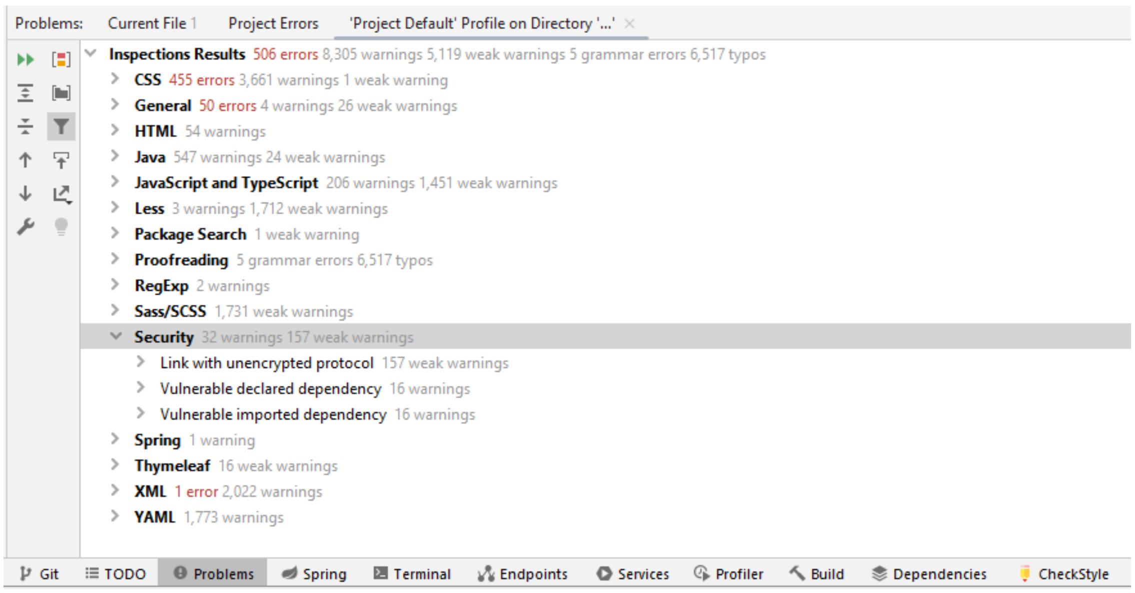
```
Rules: <active configuration>
> result.java : 41 item(s)
> result_count.java : 19 item(s)
> results.java : 10 item(s)
> scan.java : 4 item(s)
v severities.java : 8 item(s)
    Name 'severities' must match pattern '^[A-Z][a-zA-Z0-9]*$'. (5:14) [TypeName]
    File contains tab characters (this is the first instance). (6:1) [FileTabCharacter]
    Missing a Javadoc comment. (6:9) [JavadocVariable]
    Variable 'severity' must be private and have accessor methods. (6:25) [VisibilityModifier]
    Missing a Javadoc comment. (7:9) [JavadocVariable]
    Variable 'score' must be private and have accessor methods. (7:21) [VisibilityModifier]
    Missing a Javadoc comment. (8:9) [JavadocVariable]
    Variable 'text' must be private and have accessor methods. (8:23) [VisibilityModifier]
> severity.java : 18 item(s)
> solution.java : 6 item(s)
> sort.java : 4 item(s)
```

Figure 4: CheckStyle report for "severities.java" file

After installing the corresponding plugin, the IDE must be restarted. The next step is to right-click on the name of the target project and choose "Analyze" → "Inspect code". As it is depicted in Figure 4, in the corresponding "CheckStyle" tap, in the IntelliJ user interface, error messages are presented for each individual file. In our specific example, "severities.java" seems to be associated with 8 such errors. The name of the file does not match the default type patterns, unnecessary tab characters are included in the file, a Javadoc comment is missing in three different locations, and finally, three of the corresponding Java class variables are characterized as public, although they should be private. There is a large set of files that is mentioned in the CheckStyle's feedback. However, the variety of the errors is not large. Same errors are mentioned in most of the included files.

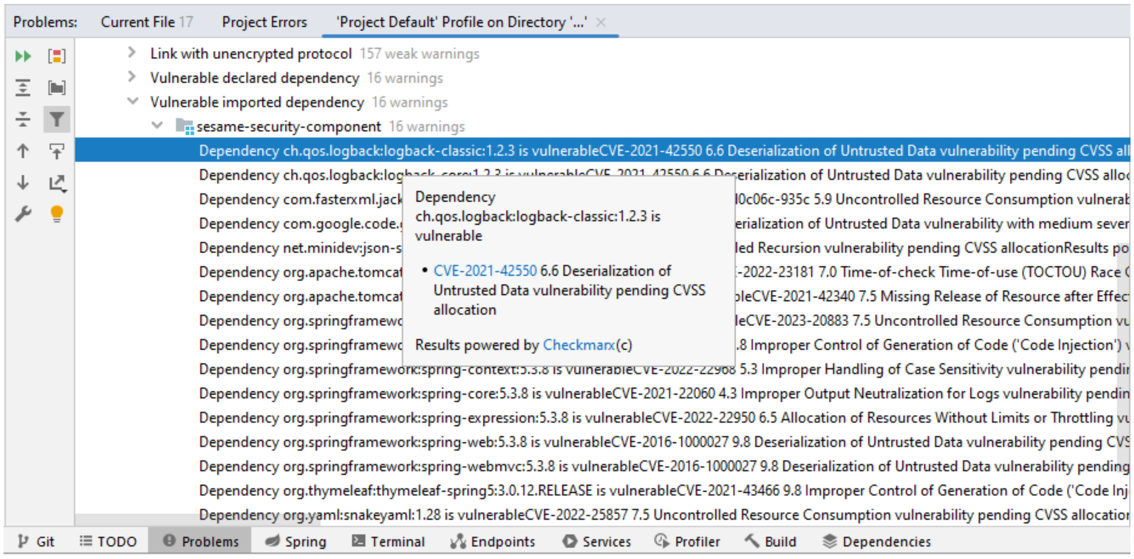
3.1.1.2 *IntelliJ's native code analyser*

Since we started the description of the code analysers with those that can be integrated with IDEs, we should continue with the native code analyser of IntelliJ. Asking for code inspection of a project, a report is created in the corresponding "Problems" tab. This time the reported errors are not limited to documentation. As it can be seen in Figure 5, a great number of errors, warnings, weak warnings, grammar errors, and typos are mentioned. What is really interesting is the fact that some of them are related to known vulnerabilities.


Problems: Current File 1 Project Errors 'Project Default' Profile on Directory '...'
Git TODO Problems Spring Terminal Endpoints Services Profiler Build Dependencies CheckStyle

```
Inspections Results 506 errors 8,305 warnings 5,119 weak warnings 5 grammar errors 6,517 typos
  CSS 455 errors 3,661 warnings 1 weak warning
  General 50 errors 4 warnings 26 weak warnings
  HTML 54 warnings
  Java 547 warnings 24 weak warnings
  JavaScript and TypeScript 206 warnings 1,451 weak warnings
  Less 3 warnings 1,712 weak warnings
  Package Search 1 weak warning
  Proofreading 5 grammar errors 6,517 typos
  RegExp 2 warnings
  Sass/SCSS 1,731 weak warnings
  Security 32 warnings 157 weak warnings
    Link with unencrypted protocol 157 weak warnings
    Vulnerable declared dependency 16 warnings
    Vulnerable imported dependency 16 warnings
  Spring 1 warning
  Thymeleaf 16 weak warnings
  XML 1 error 2,022 warnings
  YAML 1,773 warnings
```

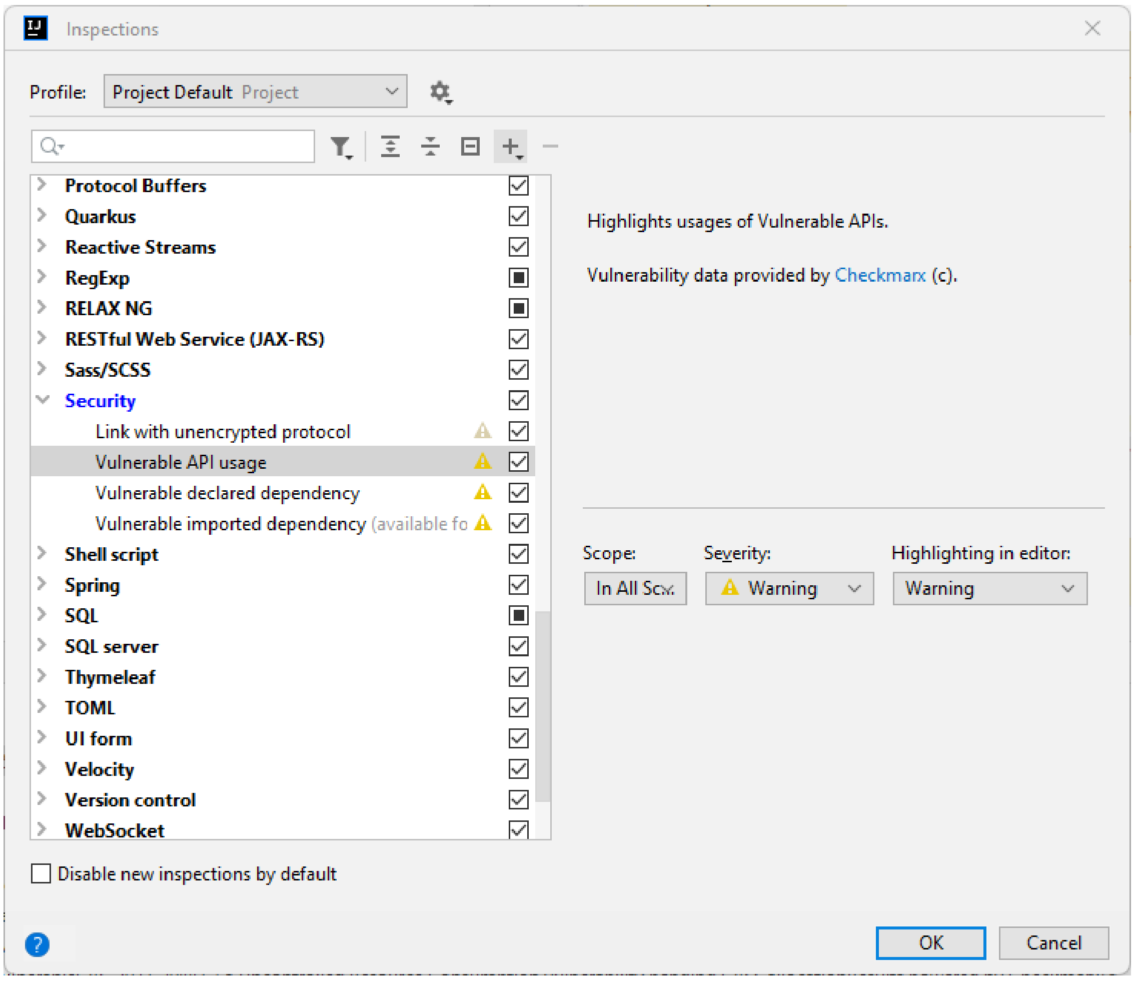
Figure 5: Report from the IntelliJ's native code analyzer (checkmarx)



```
Problems:   Current File 17   Project Errors   'Project Default' Profile on Directory '...'

  > Link with unencrypted protocol 157 weak warnings
  > Vulnerable declared dependency 16 warnings
  v Vulnerable imported dependency 16 warnings
      v sesame-security-component 16 warnings
          Dependency ch.qos.logback:logback-classic:1.2.3 is vulnerableCVE-2021-42550 6.6 Deserialization of Untrusted Data vulnerability pending CVSS all
          Dependency ch.qos.logback:log                    erialization of Untrusted Data vulnerability pending CVSS alloc
          Dependency com.fasterxml.jack                    l0c06c-935c 5.9 Uncontrolled Resource Consumption vulnerab
          Dependency com.google.code.                      erialization of Untrusted Data vulnerability with medium sever
          Dependency net.minidev:json-s                    led Recursion vulnerability pending CVSS allocationResults po
          Dependency org.apache.tomcat                     -2022-23181 7.0 Time-of-check Time-of-use (TOCTOU) Race C
          Dependency org.apache.tomcat                     bleCVE-2021-42340 7.5 Missing Release of Resource after Effec
          Dependency org.springframewo                     leCVE-2023-20883 7.5 Uncontrolled Resource Consumption vu
          Dependency org.springframewo                     .8 Improper Control of Generation of Code ('Code Injection') v
          Dependency org.springframework:spring-context:5.3.8 is vulnerableCVE-2022-22968 5.3 Improper Handling of Case Sensitivity vulnerability pendin
          Dependency org.springframework:spring-core:5.3.8 is vulnerableCVE-2021-22060 4.3 Improper Output Neutralization for Logs vulnerability pendin
          Dependency org.springframework:spring-expression:5.3.8 is vulnerableCVE-2022-22950 6.5 Allocation of Resources Without Limits or Throttling vu
          Dependency org.springframework:spring-web:5.3.8 is vulnerableCVE-2016-1000027 9.8 Deserialization of Untrusted Data vulnerability pending CVS
          Dependency org.springframework:spring-webmvc:5.3.8 is vulnerableCVE-2016-1000027 9.8 Deserialization of Untrusted Data vulnerability pending
          Dependency org.thymeleaf:thymeleaf-spring5:3.0.12.RELEASE is vulnerableCVE-2021-43466 9.8 Improper Control of Generation of Code ('Code Inj
          Dependency org.yaml:snakeyaml:1.28 is vulnerableCVE-2022-25857 7.5 Uncontrolled Resource Consumption vulnerability pending CVSS allocatio

    Dependency
    ch.qos.logback:logback-classic:1.2.3 is
    vulnerable

    • CVE-2021-42550 6.6 Deserialization of
      Untrusted Data vulnerability pending CVSS
      allocation

    Results powered by Checkmarx(c)

Git   TODO   Problems   Spring   Terminal   Endpoints   Services   Profiler   Build   Dependencies
```

Figure 6: Security warning from the IntelliJ's native code analyzer (checkmarx)

This is the case for the "security" group of warnings (32 warnings and 157 weak warnings are mentioned there). Figure 6 dives into one of them, under the "Vulnerable imported dependency" categorization. It seems that one of the dependencies of our code is ch.qos.logbacklogback-classic:1.2.3 which is characterized as vulnerable and is associated with CVE-2021-42550 known vulnerability. According to the National Vulnerability Database (NIST), "In logback version 1.2.7 and prior versions, an attacker with the required privileges to edit configurations files could craft a malicious configuration allowing to execute arbitrary code loaded from LDAP servers".


Inspections

Profile: Project Default Project

- Protocol Buffers ☑
- Quarkus ☑
- Reactive Streams ☑
- RegExp ■
- RELAX NG ■
- RESTful Web Service (JAX-RS) ☑
- Sass/SCSS ☑
- Security ☑
  - Link with unencrypted protocol ⚠ ☑
  - Vulnerable API usage ⚠ ☑
  - Vulnerable declared dependency ⚠ ☑
  - Vulnerable imported dependency (available fo ⚠ ☑
- Shell script ☑
- Spring ☑
- SQL ■
- SQL server ☑
- Thymeleaf ☑
- TOML ☑
- UI form ☑
- Velocity ☑
- Version control ☑
- WebSocket ☑

Highlights usages of Vulnerable APIs.

Vulnerability data provided by Checkmarx (c).

Scope: In All Scx. Severity: ⚠ Warning Highlighting in editor: Warning

☐ Disable new inspections by default

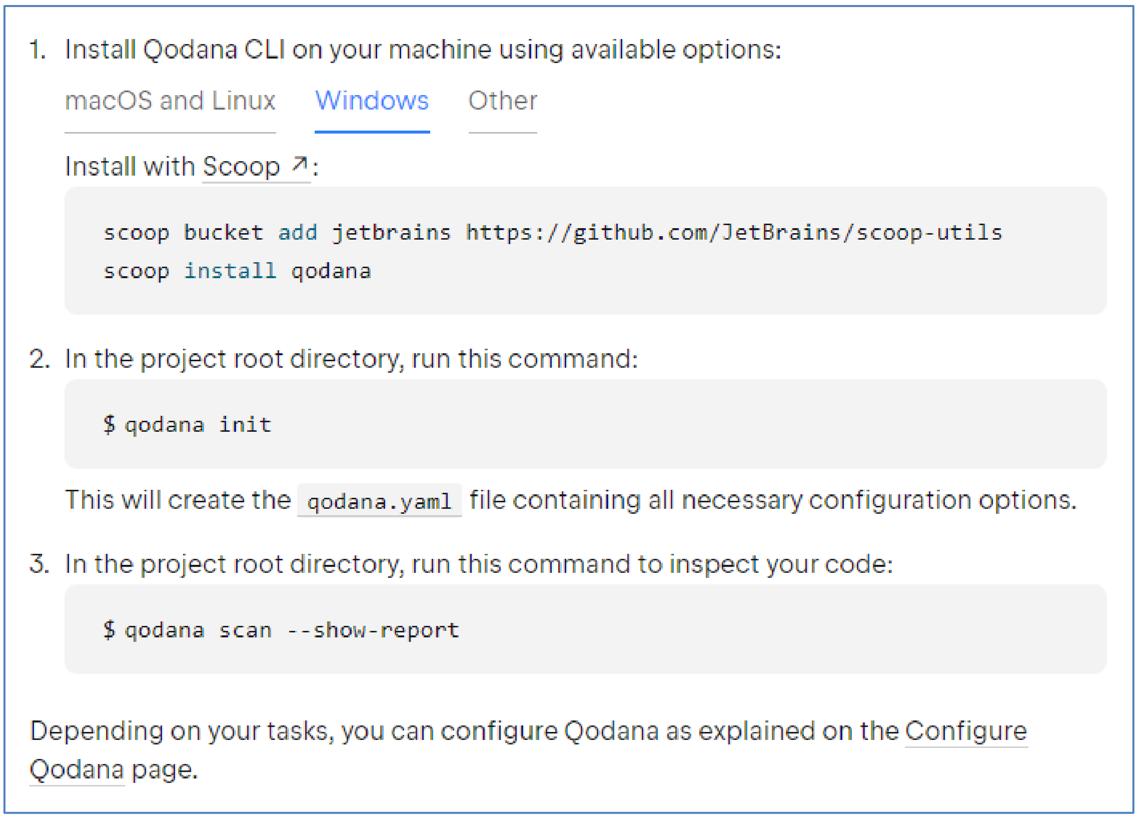
Figure 7: Configuration of the IntelliJ's native code analyzer (checkmarx)

Figure 7 shows how the produced report can be configured, including or excluding report categorization groups. Moreover, the scope of the scan can be decided, the severity of the detected issues, and which of them should be highlighted in the editor.

3.1.1.3 Qodana

Qodana is a static code analysis tool developed by JetBrains, which leverages advanced static analysis techniques to detect potential issues, bugs, vulnerabilities, and code smells in various programming languages. Its techniques include pre-defined rules along with machine learning algorithms.

The installation instructions for Windows are presented in Figure 8. Then installation is done with Scoop. After downloading the code, the creation of the qodana.yaml configuration file follows. The scan command starts the code analysis process.



1. Install Qodana CLI on your machine using available options:

   macOS and Linux | Windows | Other

   Install with Scoop ↗:

```
scoop bucket add jetbrains https://github.com/JetBrains/scoop-utils
scoop install qodana
```

2. In the project root directory, run this command:

```
$ qodana init
```

   This will create the `qodana.yaml` file containing all necessary configuration options.

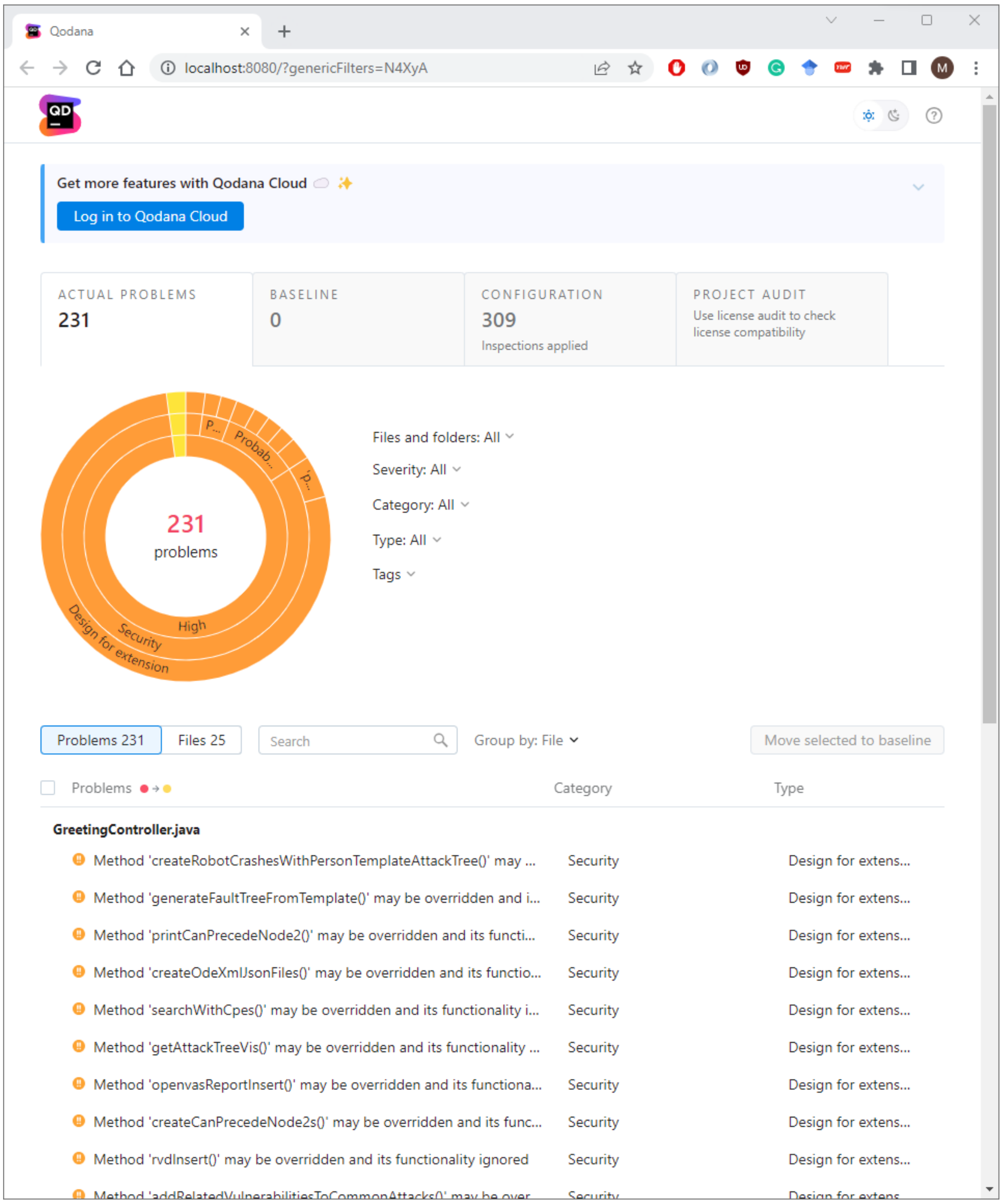
3. In the project root directory, run this command to inspect your code:

```
$ qodana scan --show-report
```

Depending on your tasks, you can configure Qodana as explained on the Configure Qodana page.

Figure 8: Installation instructions for Qodana

By the end of the code analysis process, a web user interface is made available (Figure 9). Under the "Actual problems" tab the number of the identified problems is mentioned along with a number of categorization names. In this specific example, the majority of the reported problems are considered of high severity, with a large subset of them to be categorised as security problems.


Qodana

localhost:8080/?genericFilters=N4XyA

Get more features with Qodana Cloud

Log in to Qodana Cloud

ACTUAL PROBLEMS	BASELINE	CONFIGURATION	PROJECT AUDIT
231	0	309 Inspections applied	Use license audit to check license compatibility

Files and folders: All

Severity: All

Category: All

Type: All

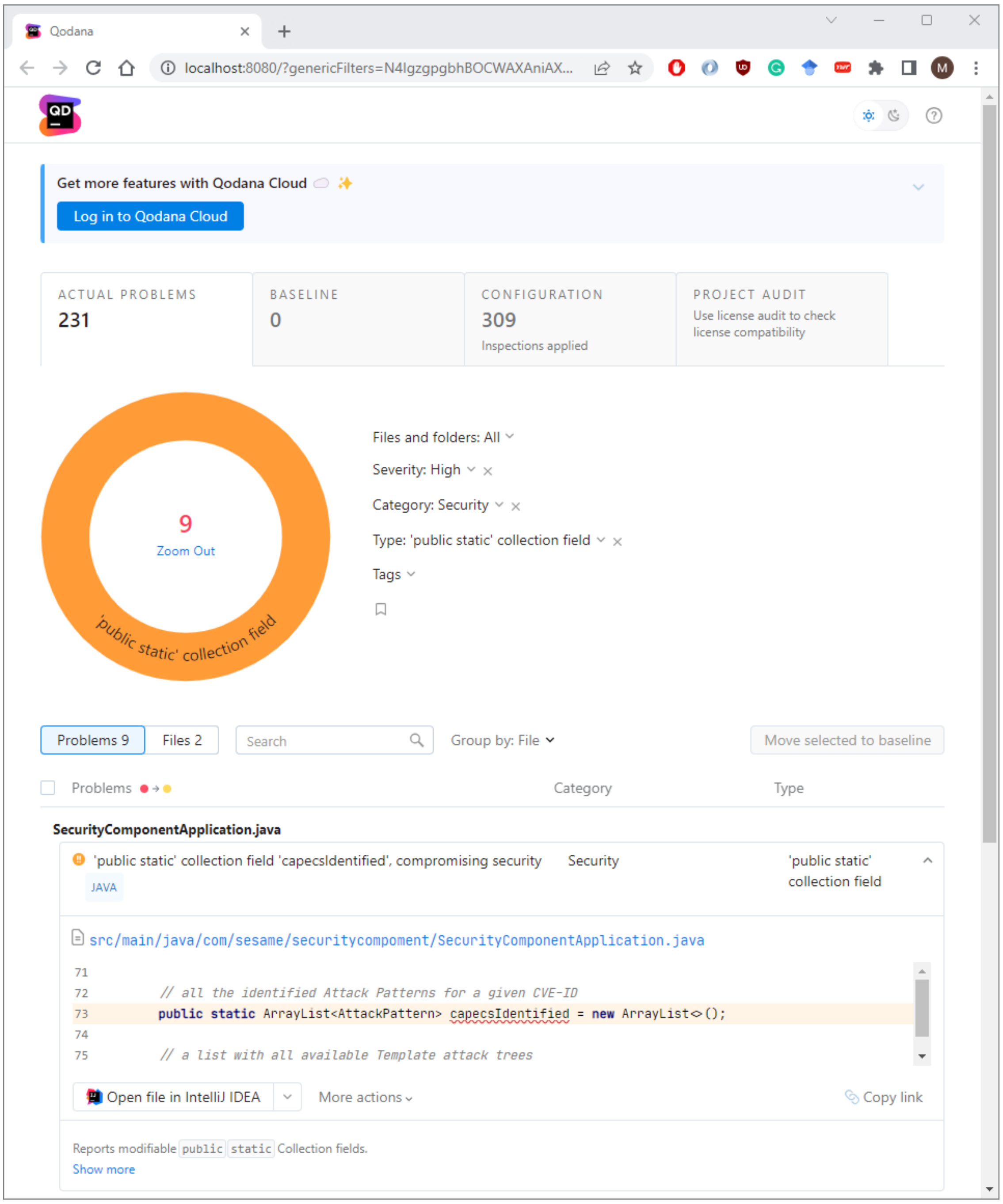
Tags

Problems 231 | Files 25 | Search | Group by: File | Move selected to baseline

Problems	Category	Type
GreetingController.java		
Method 'createRobotCrashesWithPersonTemplateAttackTree()' may ...	Security	Design for extens...
Method 'generateFaultTreeFromTemplate()' may be overridden and i...	Security	Design for extens...
Method 'printCanPrecedeNode2()' may be overridden and its functi...	Security	Design for extens...
Method 'createOdeXmlJsonFiles()' may be overridden and its functio...	Security	Design for extens...
Method 'searchWithCpes()' may be overridden and its functionality i...	Security	Design for extens...
Method 'getAttackTreeVis()' may be overridden and its functionality ...	Security	Design for extens...
Method 'openvasReportInsert()' may be overridden and its functiona...	Security	Design for extens...
Method 'createCanPrecedeNode2s()' may be overridden and its func...	Security	Design for extens...
Method 'rvdInsert()' may be overridden and its functionality ignored	Security	Design for extens...
Method 'addRelatedVulnerabilitiesToCommonAttacks()' may be over	Security	Design for extens

Figure 9: Qodana web user interface

Figure 10 presents the detailed report of one of the identified code problems. The problem was detected in the "SecurityComponentApplication.java" file, is called "'public static' collection field 'capecsIdentified', compromizsing security", is categorised as a security problem, and its type is "'public static' collection field". According to the Common Weakness Enumerator and the CWE-582: Array Declared Public, Final, and Static specific weakness, declaring an array public, final, and static, is not sufficient to prevent the array's contents from being modified. Although the "final" Java constraint requires that the array object itself is assigned only once, it cannot guarantee that no changes will happen on the values of the array elements, compromizing the integrity of the application data.


Qodana
localhost:8080/?genericFilters=N4lgzgpgbhBOCWAXAniAX...
Get more features with Qodana Cloud
Log in to Qodana Cloud
Files and folders: All
Severity: High
Category: Security
Type: 'public static' collection field
Tags
9
Zoom Out
'public static' collection field
Problems 9
Files 2
Search
Group by: File
Move selected to baseline
Open file in IntelliJ IDEA
More actions
Copy link
Show more

ACTUAL PROBLEMS	BASELINE	CONFIGURATION	PROJECT AUDIT
231	0	309 Inspections applied	Use license audit to check license compatibility

Problems	Category	Type
SecurityComponentApplication.java		
'public static' collection field 'capecsIdentified', compromising security JAVA	Security	'public static' collection field

src/main/java/com/sesame/securitycompoment/SecurityComponentApplication.java

```

        // all the identified Attack Patterns for a given CVE-ID
        public static ArrayList<AttackPattern> capecsIdentified = new ArrayList<>();

        // a list with all available Template attack trees
```

Reports modifiable public static Collection fields.

Figure 10: "public static" collection field "capecsIdentified" ", compromizsing security problem

3.1.1.4 SonarQube

SonarQube is yet another static analysis tool that we chose to use mostly due to its popularity and range of supported languages. SonarQube offers static code analysis capabilities for various programming languages, including Python, and Python is the language that is used for the development of safety related parts of EDDIs. Moreover, another characteristic that we wanted to demonstrate is the fact that it can be connected with GitHub and other code hosting platforms for collaboration and version control. SESAME already uses GitHub as a repository for miscellaneous project related information, including code. Putting the code of all the created EDDIs under an organisation and linking that organisation with SonarQube, seems a relatively easy way to perform remote static analysis to the executable dependability decryptions of the SESAME use cases.

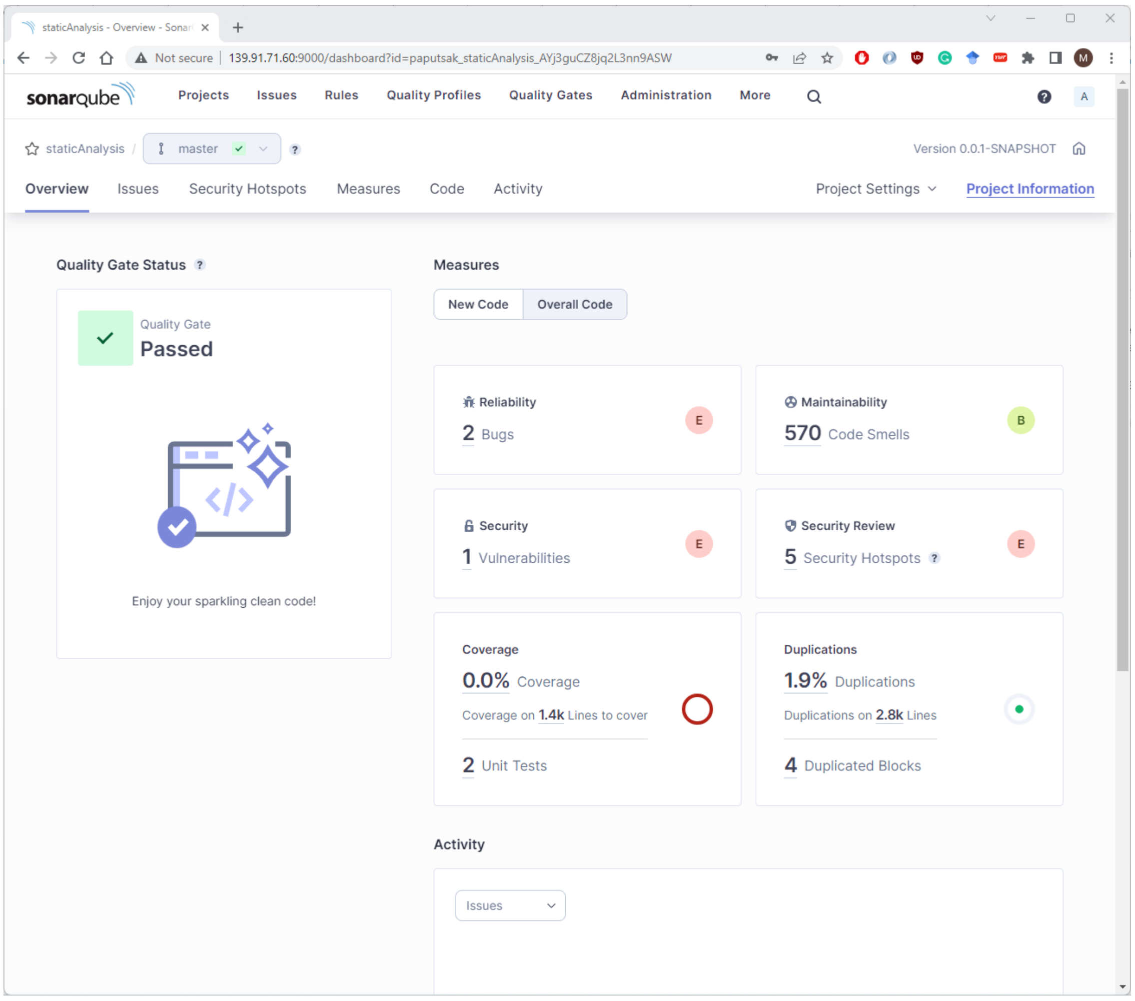
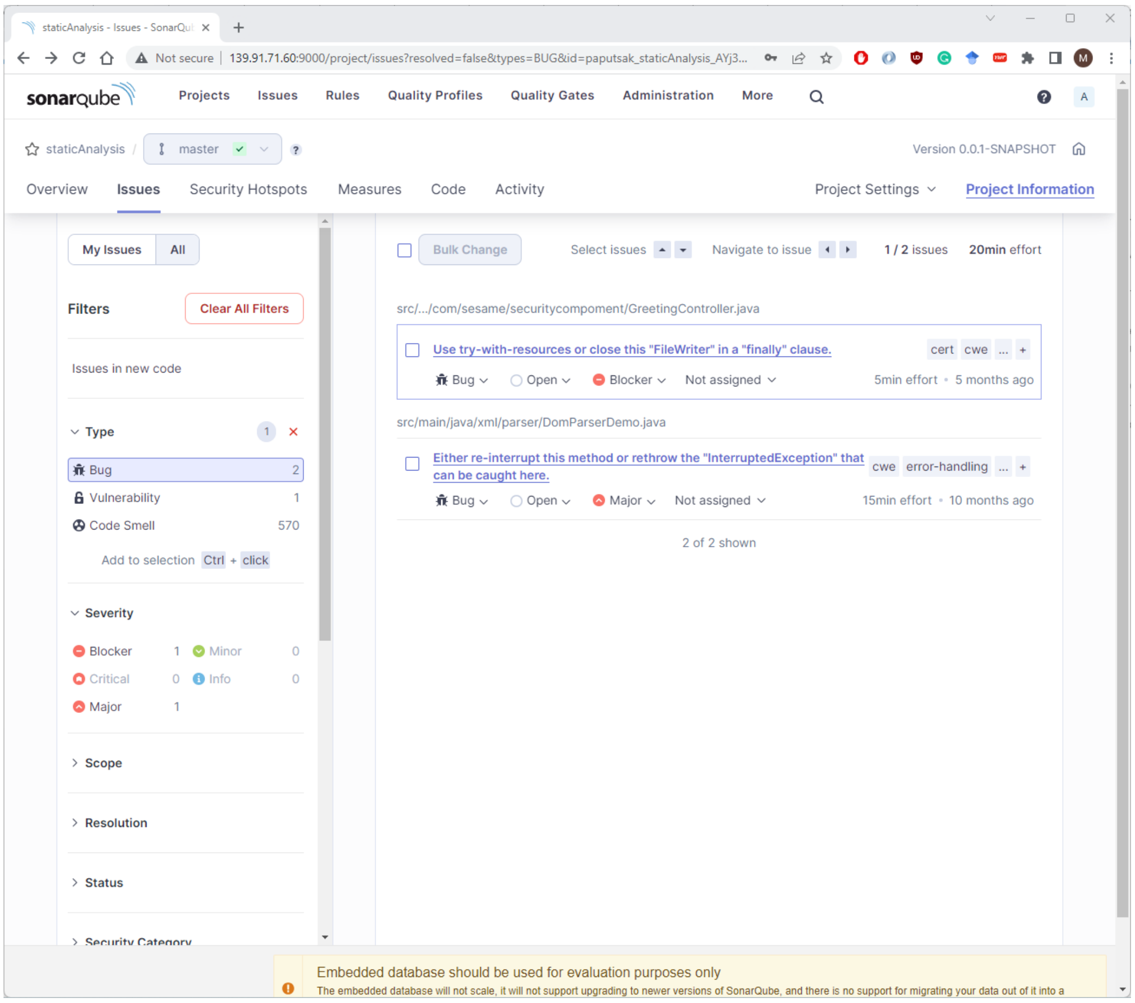


Figure 11: SonarQube web available dashboard

Figure 11 depicts the welcome page of the SonarQube dashboard. At the top left side of the page we see that SonarQube is connected with the "staticAnalysis" project and the "master" branch. According to the overview that is depicted in this figure, the "Quality Gate Status" is marked as "passed". Moreover, 2 Bugs, 570 Code Smells, 1 Vulnerability, and 5 Security Hotspots are detected.

Moving to the "Issues" tab, we can see more details about all the reported problems. In Figure 12, we have selected the Bugs from the left side menu. Details about the two identified Bugs are presented in the right panel. The first one is located in the "GreetingController.java" file and is associated with the "FileWriter" object. The second is located in the "DomParserDemo.java" file and is associated with the catching of the "InterruptedException" exception.


staticAnalysis - Issues - SonarQu

Not secure | 139.91.71.60:9000/project/issues?resolved=false&types=BUG&id=paputsak_staticAnalysis_AYj3...

sonarqube
Projects Issues Rules Quality Profiles Quality Gates Administration More
staticAnalysis / master
Version 0.0.1-SNAPSHOT
Overview Issues Security Hotspots Measures Code Activity Project Settings Project Information
My Issues All
Filters Clear All Filters
Issues in new code
Type 1
Bug 2
Vulnerability 1
Code Smell 570
Add to selection Ctrl + click
Severity
Blocker 1 Minor 0
Critical 0 Info 0
Major 1
Scope
Resolution
Status
Security Category
Bulk Change Select issues Navigate to issue 1 / 2 issues 20min effort

src/.../com/sesame/securitycompoment/GreetingController.java

Use try-with-resources or close this "FileWriter" in a "finally" clause. cert cwe ...
Bug Open Blocker Not assigned 5min effort • 5 months ago

src/main/java/xml/parser/DomParserDemo.java

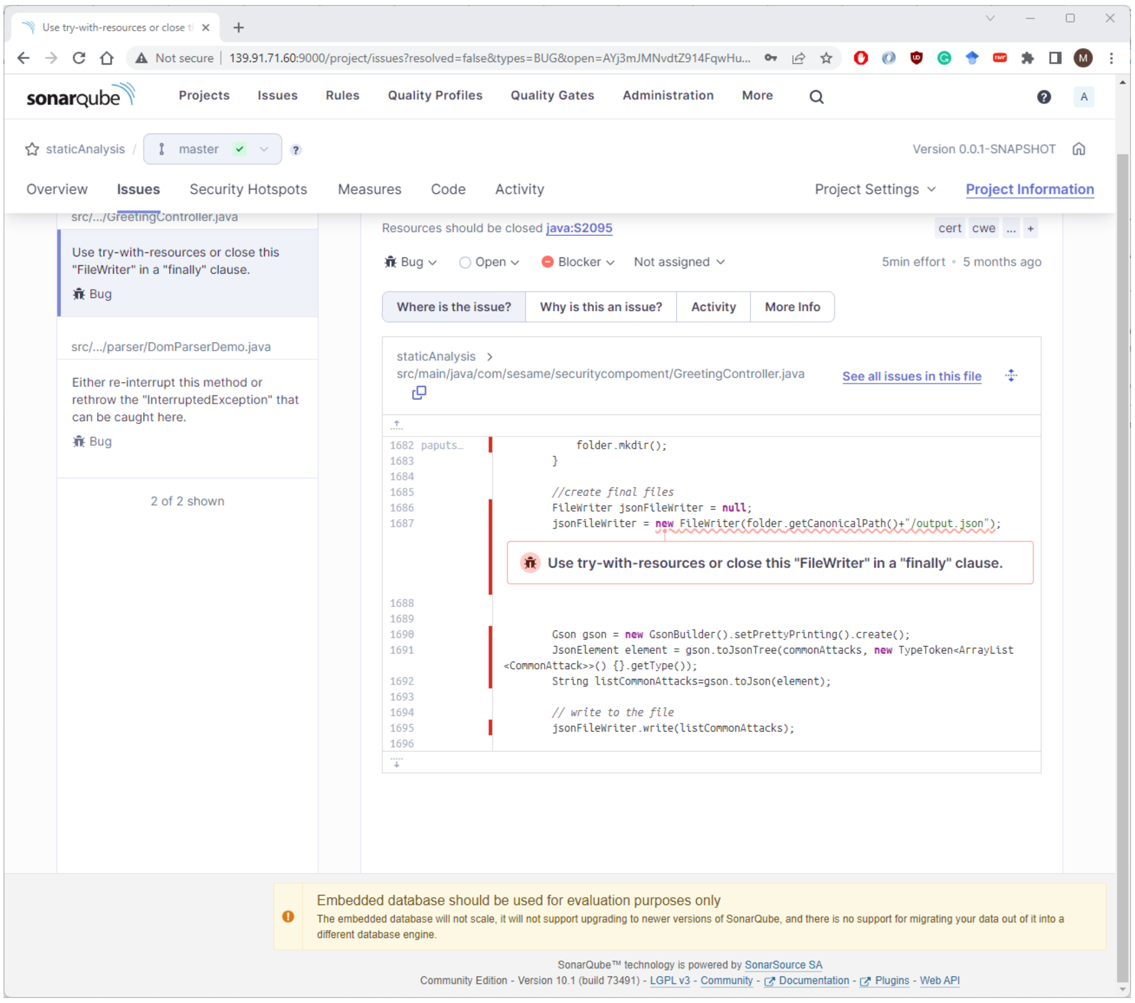
Either re-interrupt this method or rethrow the "InterruptedException" that can be caught here. cwe error-handling ...
Bug Open Major Not assigned 15min effort • 10 months ago

2 of 2 shown

Embedded database should be used for evaluation purposes only
The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a

Figure 12: SonarQube reported Bugs

We click on the first one, trying to reveal more information about the Bug. Figure 13 shows the page we are driven to. What is clear now, is at which line of code this Bug is detected, under the “Where is the issue?” tab. The “Why is this an issue?” tab, depicted in Figure 14, has even more information regarding the issue itself. The issue in this example is that we a “FileWriter” object to read a file, but we never close it. In fact, the “close” call is included in the code but not in a “finally” block as SonarQube indicates. Additional information is available regarding the consequences of such an issue. An application that does not properly close resources, will cause a resource leak, making the application itself and the corresponding host to struggle.


Use try-with-resources or close t
139.91.71.60:9000/project/issues?resolved=false&types=BUG&open=AYj3mJMNvdtZ914FqwHu...
sonarqube
Projects
Issues
Rules
Quality Profiles
Quality Gates
Administration
More
staticAnalysis
master
Version 0.0.1-SNAPSHOT
Overview
Issues
Security Hotspots
Measures
Code
Activity
Project Settings
Project Information
src/.../GreetingController.java
src/.../parser/DomParserDemo.java
2 of 2 shown
Resources should be closed java:S2095
cert
cwe
Bug
Open
Blocker
Not assigned
5min effort · 5 months ago
Where is the issue?
Why is this an issue?
Activity
More Info
staticAnalysis >
src/main/java/com/sesame/securitycompoment/GreetingController.java
See all issues in this file

Use try-with-resources or close this "FileWriter" in a "finally" clause.
Bug

Either re-interrupt this method or rethrow the "InterruptedException" that can be caught here.
Bug

```
            folder.mkdir();
        }

        //create final files
        FileWriter jsonFileWriter = null;
        jsonFileWriter = new FileWriter(folder.getCanonicalPath()+"/output.json");
```

Use try-with-resources or close this "FileWriter" in a "finally" clause.

```


        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        JsonElement element = gson.toJsonTree(commonAttacks, new TypeToken<ArrayList
<CommonAttack>>() {}.getType());
        String listCommonAttacks=gson.toJson(element);

        // write to the file
        jsonFileWriter.write(listCommonAttacks);

```

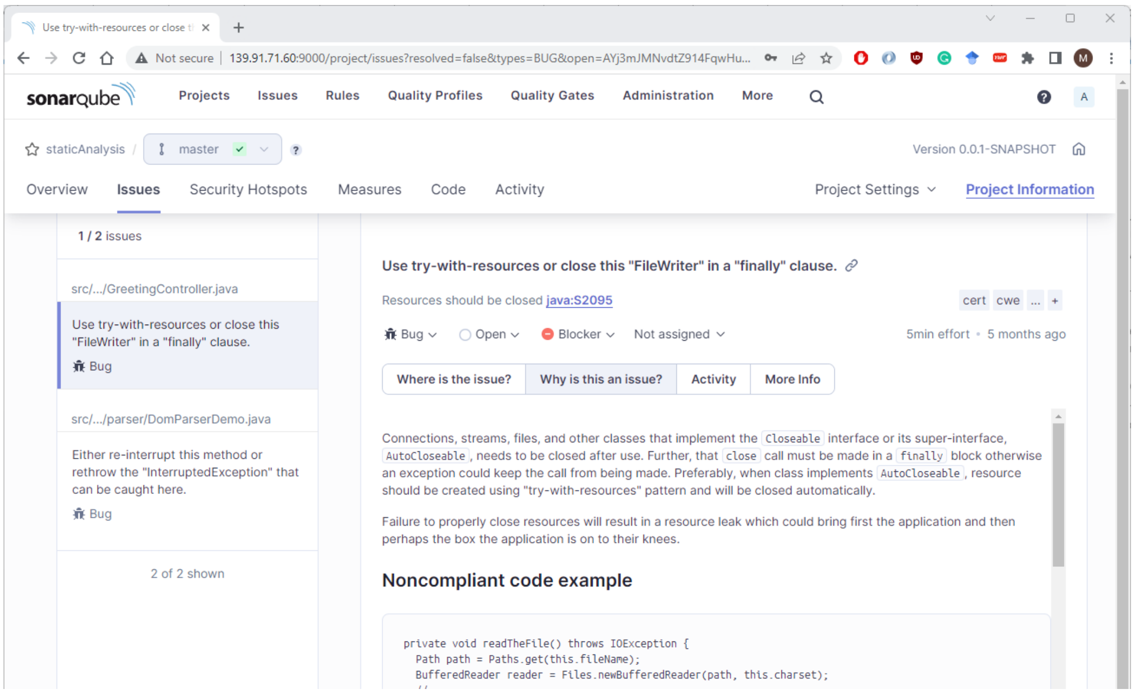
Embedded database should be used for evaluation purposes only
The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.

SonarQube™ technology is powered by SonarSource SA
Community Edition - Version 10.1 (build 73491) - LGPL v3 - Community - Documentation - Plugins - Web API

Figure 13: SonarQube - "Where is the issue?" tab

Finally, the “More Info” tab show the resources of the information that we saw in the previous tabs. What is very interesting here is that among the resources the Common Weakness Enumeration is included, linking the reported issues with known weaknesses.

In our example, the code issue of the “FileWriter” is associated with CWE-459 and CWE-772. According to the description of the CWE-459: Incomplete Cleanup weakness, an application may not remove temporary or supporting resources after they have been used. This is a not language-specific weakness that could lead to overflow of the number of temporary files and create a denial of service problem. As a detection method, automated static analysis is mentioned with high effectiveness. The second weakness, CWE-772: Missing Release of Resources after Effective Lifetime, is similar to the first one. According to its description, the allocation of resources without releasing them can allow attackers to cause denial of service.



Use try-with-resources or close this "FileWriter" in a "finally" clause.

Either re-interrupt this method or rethrow the "InterruptedException" that can be caught here.

Use try-with-resources or close this "FileWriter" in a "finally" clause.

Resources should be closed java:S2095

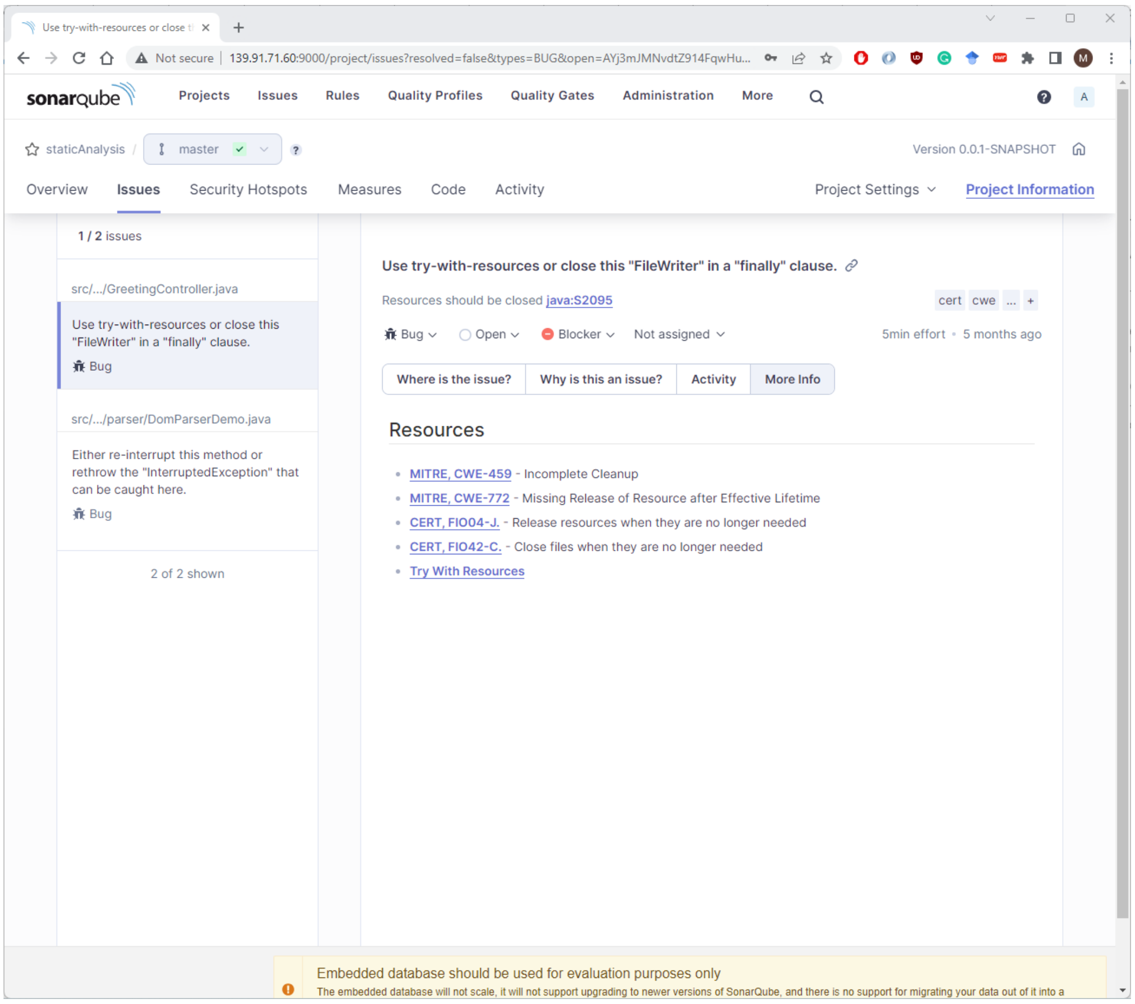
Connections, streams, files, and other classes that implement the `Closeable` interface or its super-interface, `AutoCloseable`, needs to be closed after use. Further, that `close` call must be made in a `finally` block otherwise an exception could keep the call from being made. Preferably, when class implements `AutoCloseable`, resource should be created using "try-with-resources" pattern and will be closed automatically.

Failure to properly close resources will result in a resource leak which could bring first the application and then perhaps the box the application is on to their knees.

Noncompliant code example

```
private void readTheFile() throws IOException {
  Path path = Paths.get(this.fileName);
  BufferedReader reader = Files.newBufferedReader(path, this.charset);
```

Figure 14: SonarQube - "Why is this an issue?" tab



Use try-with-resources or close this "FileWriter" in a "finally" clause.

Either re-interrupt this method or rethrow the "InterruptedException" that can be caught here.

Use try-with-resources or close this "FileWriter" in a "finally" clause.

Resources should be closed java:S2095

Resources

- MITRE, CWE-459 - Incomplete Cleanup
- MITRE, CWE-772 - Missing Release of Resource after Effective Lifetime
- CERT, FIO04-J. - Release resources when they are no longer needed
- CERT, FIO42-C. - Close files when they are no longer needed
- Try With Resources

Embedded database should be used for evaluation purposes only

The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a

Figure 15: SonarQube - "More info" tab

3.1.2 Hash verification

There is a great number of tools that allow for creating a hash value for a given file and even compare hash values for integrity checking; CertUtil, HashCheck, GNU Core Utilities, HashTab, OpenSSL, QuickHash, Hasher, HashMyFiles, WinHasher, PapidCRC, HashCheckup, GtkHash, Hashdeep, OpenHashTab, and Hashrat just to name a few. In this non-exhaustive list there are tools that i) offer just a command-line interface or a user-friendly graphical interface; ii) are dedicated to a specific operating system or offer cross-platform adoption; iii) focus on specific hash algorithms or support a wide range of them; iv) can integrate their result in the file properties dialog or not; and v) offer just hash values generation or additional features such as file comparison and batch processing.

This plethora of hash verification tools offers many choices to the end users based on concepts such as platform compatibility, UI preferences, tool functionality, tool development communities, and sector specialization. Platform compatibility defines the operating system a tool is created for. Users could choose a tool that is compatible with their preferred operating system. UI preferences include choices such as command-line for simplicity and flexibility and graphical interface for user-friendly experience. Tool functionality allows focusing on specific tool features such as a wide range of hash algorithms, recursive hashing of directories, or integration with file managers. Tool development communities is another reason for choosing (or rejecting) a tool. An active community could offer an up-to-date documentation of the tool or even online support though forums and direct communication. Finally, sector specialization is another characteristic that differentiates tools. In the case of hash verification, there are hash verification tools tailored for digital forensics, software development, or cybersecurity purposes.

Towards fulfilling SESAME needs for hash verification we tested some off-the-shelf tools dedicated to different operating systems. However, we have taken the initiative to develop our own version of a hash verification tool using Python. By doing so, we have more control on the overall verification process and we are able to create a highly customizable tool, which can be easily adopted from the individual SESAME use cases. Moreover, creating our own tool, enhanced our understanding and familiarization with the hash creation and verification process. Another reason for developing our own tool is our intention to update and enhance it whenever it is necessary to keep up with security standards and newly discovered software vulnerabilities.

Figure 16 presents how such a tool could be used in an EDDI development/deployment workflow. The goal is the verification to be executed on the robot just before the execution of an EDDI. In that way, we will be sure that the EDDI to be executed is not tampered with in any way.

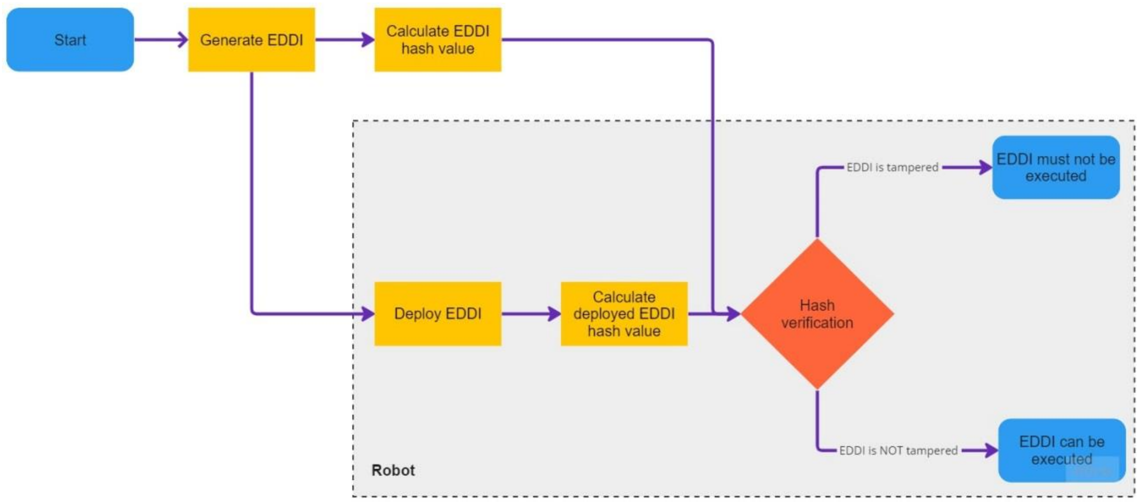


Figure 16: Hash Verification tool in the EDDI workflow

Listing 1 below presents a Python script with a function that calculates a hash value for a given file (calculate_hash), and a second one that compares pairs of hash values deciding if they match or not (compare_hash). The “calculate_hash()” function accepts two parameters, a path to a target file and a list of hash algorithms. It reads the file in binary code and in chunks to handle large files in an efficient way. The chunks are 4096 bytes in size. It then calculates a hash value for each of the given algorithms (hash_object.update), retrieving the hexadecimal representation of it (hash_object.hexdigest). At the last lines of the function, the hash values are stored in a dictionary where keys are the algorithm names and values the calculated hash values. The “file_path” variable is used for defining the path of the file to be used for the creation of the hash values, while the desired hash algorithms are mentioned in the “hash_algorithms” list. According to the presented implementation, the desired hash algorithms must be supported by the “hashlib” module in Python, part of the Python Standard Library. It provides various hash functions for calculating hash values for string or binary data, such as MD5, SHA-1, and SHA-256. The functions that “hashlib” module offers include “new()” to create a hash object for a specific algorithm, “update()” method to feed data into the created hash object, and “hexdigest()” to retrieve the hash value in a hexadecimal format, readable by humans. Moreover, these “hashlib” functions are a consistent way to interact with the available hash algorithms, independently of which algorithm you choose.

```python
import hashlib

def calculate_hash(file_path, hash_algorithms):

    hash_values = {}


    # Read the file in binary mode

    with open(file_path, 'rb') as file:

        while True:

            # Read the file in chunks to handle large files efficiently

            chunk = file.read(4096)
            if not chunk:
                break

            # Calculate hash values for each specified algorithm
            for algorithm in hash_algorithms:
                # Create a hash object
                hash_object = hashlib.new(algorithm)
                hash_object.update(chunk)

                # Get the hexadecimal representation of the hash value
                hash_value = hash_object.hexdigest()

                # Store the hash value for the algorithm
                if algorithm not in hash_values:
                    hash_values[algorithm] = hash_value
                else:
                    hash_values[algorithm] += hash_value

    return hash_values

def compare_hash(hash_values, pre_calculated_values):
    for algorithm, hash_value in hash_values.items():
        if algorithm in pre_calculated_values:
            if hash_value == pre_calculated_values[algorithm]:
                print(f"{algorithm}: Hash value matches the pre-defined val-
ue.")
            else:
                print(f"{algorithm}: Hash value does not match the pre-
defined value.")
        else:
            print(f"{algorithm}: Pre-defined value not found for compari-
son.")

# Specify the file path and hash algorithms to use
file_path = 'path/to/file.txt'
```

```python
hash_algorithms = ['md5', 'sha1', 'sha256']


# Calculate the hash values for the file using the specified algorithms
hash_values = calculate_hash(file_path, hash_algorithms)


# Define the pre-defined hash values for comparison
pre_calculated_values = {
    'md5': '...',
    'sha1': '...',
    'sha256': '...',
}


# Compare the calculated hash values with the pre-defined values
compare_hash(hash_values, pre_calculated_values)
```

Listing 1: Hash verification program in Python

The “compare_hash()” function takes the calculated hash values (hash_values) and a dictionary of calculated at some point in the past hash values (pre_calculated_values) as input. The “pre_calculated_values” are hash values that have be produced for individual files, most probably during their creation or after a formation of a new version of them. These values correspond to hash values calculated during the creation of EDDIs, in the context of SESAME. The “compare_hash()” function compares each calculated hash value with the corresponding pre-defined value and prints whether they match or not.

The “pre_calculated_values” of Listing 1 were stored in a dictionary. However, the goal is, values like these to be stored in a central location, and to be accessed every time the “compare_hash” needs to be called. Listing 2 below, is a REST API in Python, which reads hash values from an SQLite database and serves them. If the database does not exist, it creates a new one.

The “create_database()” function is responsible for creating a SQLite database and a table within it. Inside the “create_database()”, the “sqlite3.connect()” function establishes a connection to the database. The “DATABASE” variable defines the path or the name of the database file. The “sqlite3.connect()” function returns a connection object (“conn”). The next code line creates a cursor object that is used to execute SQL statements and interact with the database.

An SQL element is then executed utilizing the cursor’s “execute()” method. The SQL statement creates a table named "hashes" if it doesn't already exist. The table has two columns: "algorithm" and "hash_value". The "algorithm" column is designated as the primary key of the table.

What follows is calling of “conn.commit()” method to commit the changes and save them permanently in the database. The very last thing is to close the created connection object (conn.close).

The “populate_database()” function populates the SQLite database, created in the “create_database()” function, with sample hash values. Once again, the “sqlite3.connect()” function establishes a connection to the database, returning a connection object. A cursor is also created.

After that, a dictionary named “reference_hashes” is created. This dictionary includes sample hash values for different algorithms. The algorithm names are the dictionary keys and the hash values the corresponding dictionary values.

A loop follows then (“for” statement). This loop iterates over the items in the reference_hashes dictionary using the items() method. The goal is to insert or replace the hash values into the database. Inside the loop an SQL statement is executed with the “execute()” method. The SQL statement uses the “INSERT OR REPLACE INTO” syntax to insert a new row or replace an existing row in the "hashes" table. The algorithm name and hash value are passed as parameters using placeholders (“?”), and the actual values are provided as a tuple “(algorithm, hash_value)”.

After the insertion of the hash values, the “conn.commit()” is called to commit the changes and save them permanently in the database. Finally, the code closes the database connection using the conn.close() method.

```
from flask import Flask, jsonify
import sqlite3

app = Flask(__name__)

DATABASE = 'hashes.db'

def create_database():
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()

    # Create the table if it doesn't exist
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS hashes (
            algorithm TEXT PRIMARY KEY,
            hash_value TEXT
        )
    ''')
    conn.commit()
    conn.close()

def populate_database():
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    # Sample reference hashes
    reference_hashes = {
        'md5': '...',
        'sha1': '...',
        'sha256': '...',
        'sha512':
'da6f853676df7c1713df62e155df594fe621033170f49bf9b0a64cecac89f3f0b99d6b345c6f
5da640471775847229373a2a5dfed322bf6e882011c062359746'
    }

    # Insert or replace the reference hashes in the database
    for algorithm, hash_value in reference_hashes.items():
        cursor.execute('INSERT OR REPLACE INTO hashes (algorithm, hash_value)
VALUES (?, ?)', (algorithm, hash_value))

    conn.commit()
    conn.close()


def get_connection():
    return sqlite3.connect(DATABASE)


def get_hashes():
    conn = get_connection()
    cursor = conn.cursor()

    cursor.execute('SELECT algorithm, hash_value FROM hashes')
    rows = cursor.fetchall()
    reference_hashes = {row[0]: row[1] for row in rows}

    conn.close()
    return reference_hashes


@app.route('/api/hashes', methods=['GET'])
def hashes_route():
    reference_hashes = get_hashes()
    return jsonify(reference_hashes)


if __name__ == '__main__':
    create_database()
    populate_database()
    app.run()
```

Listing 2: REST API in Python that reads Hash values from and serves them

Finally, the “get_hashes()” function retrieves the hash values stored in the SQLite database as a dictionary. Once more a connection to the database is established and a cursor, from that connection, is created (“get_connection()”, “cursor”).

Utilizing the “execute()” method, an SQL statement is executed, which selects the "algorithm" and "hash_value" columns from the "hashes" table in the database. The “ fetchall()” method retrieves all the rows of the corresponding table. Each row represents a reference hash, where the first item (“row[0]”) is the algorithm name and the second item (“row[1]”) is the hash value.

What follows is the construction of a dictionary named “reference_hashes”. An iteration over the rows obtained from the previous step and creates key-value pairs in the dictionary, where the algorithm name is the key and the hash value is the value.

After the creation of the dictionary, the code closes the database connection using the “conn.close()” method to release the resources. Finally, the “reference_hashes” dictionary is returned, containing the retrieved reference hashes from the database.

4. CONCLUSIONS

This deliverable presents to the reader the techniques and tools that are adopted for ensuring that EDDIs remain secure despite their executable nature. A set of opensource and custom tools is described, aiming to implement methodologies such as static code analysis and hash verification.

Static code analysis allows for the source code examination towards identification of potential vulnerabilities, bugs, and security loopholes. Hash verification is the process that ensures the integrity and authenticity of executables calculating and comparing cryptographic hash values. The goal of using this technique is to detect any modifications or tampering in the executable files, providing an added layer of security.

We have adopted 5 individual static analysis tools, since each of them detects and reports different types of issues. The combination of so heterogeneous tools allows for a comprehensive solution that aims to improve code quality and eliminate chances for bugs and vulnerabilities.

Regarding hash verification, we developed our own version of a hash verification tool using Python. The goal was to create a customizable tool that could be easily adopted by the different SESAME use cases.

5. REFERENCES

[1] Louridas, Panagiotis. Static code analysis. IEEE Software 23.4, pages 58-61, 2006.

[2] Bardas, Alexandru G. Static code analysis. Journal of Information Systems & Operations Management 4.2, pages 99-107, 2010.

[3] Gomes, Ivo, et al. An overview on the static code analysis approach in software development. Faculdade de Engenharia da Universidade do Porto, Portugal, 2009.

[4] Brad Abrams Krzysztof Cwalina, Framework Design Guideline: Addison-Wesley, 2008.

[5] Novak, Jernej, and Andrej Krajnc. Taxonomy of static code analysis tools. The 33rd international convention MIPRO. IEEE, 2010.

[6] Goseva-Popstojanova, Katerina, and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. Information and Software Technology 68, pages 18-33, 2015.

[7] Mantere, Matti, Ilkka Uusitalo, and Juha Roning. Comparison of static code analysis tools. 2009 Third International Conference on Emerging Security Information, Systems and Technologies. IEEE, 2009.

[8] Lenarduzzi, Valentina, et al. A critical comparison on six static analysis tools: Detection, agreement, and precision. Journal of Systems and Software 198:111575, 2023.

[9] Anwar, Muhammad Rehan, Desy Apriani, and Irsa Rizkita Adianita. Hash Algorithm In Verification Of Certificate Data Integrity And Security. Aptisi Transactions on Technopreneurship (ATT) 3.2: pages 181-188, 2011.

[10] Goyal,V.,O'Neill,A.,&Rao,V. Correlated-input secure hash functions. *In Theory of Cryptography Conference*. Springer, Berlin, Heidelberg, pages 182-200, 2011.

[11] Lefebvre,F.,Czyz,J.,&Macq,B. A robust soft hash algorithm for digital images igntures. In Proceedings 2003 International Conference on Image Processing (Cat. No. 03CH37429), IEEE, 2003.

[12] Roy, Sujoy, and Qibin Sun. Robust hash for detecting and localizing image tampering. 2007 IEEE international conference on image processing. Vol. 6. IEEE, 2007.

[13] Chen, Yuqun, et al. Oblivious hashing: A stealthy software integrity verification primitive. Information Hiding: 5th International Workshop, IH 2002 Noordwijkerhout, The Netherlands, October 7-9, 2002 Revised Papers 5. Springer Berlin Heidelberg, 2003.

[14] Sobti, Rajeev, and Ganesan Geetha. Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)* 9.2, 2012.