POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Automatic fixing of vulnerabilities: SQL Injection in Java

Supervisor
Prof. Riccardo Sisto

Candidate
Mattia Di Leo

Reply Liquid corporate tutors
Dr. Ivan Aimale
Dr. Marco Oria
Dr.ssa Luisa Gatto

Accademic Year 2022-2023

Summary

This thesis work has been made in collaboration with the company Liquid Reply, the aim is to develop a tool capable of automatically correcting vulnerabilities within the source code.

The development of an application is a complex activity that requires a huge amount of resources. Among the various aspects, security cannot be underestimated; it is as essential as the functionalities. No matter how sophisticated and/or cutting-edge a software is, if it is vulnerable, it will still be a failure.

This aspects is usually underrated, as developers focus more on the pure logic of a program, overlooking the cybersecurity aspects due to one reason: they do not have **time**. Firstly, it would be necessary to study the fundamental of this discipline, and more important, the secured solutions are often more difficult and challenging to implement.

The tool I have developed aims to assist programmers by saving them precious time. Analyzing the state of the art, it has been observed that we are still in a nascent state. AI solutions such as ChatGPT have been discarded in favor of developing a tool that is not based on artificial intelligence.

It was chosen Java both as the development language and as the target language. Regarding vulnerabilities, SQL injection in the "vanilla" language was selected.

After developing the tool, it was initially tested to assess its effectiveness and its ability to make corrections. Finally, to optimize it further, we decided to integrate it with Git, so that at the end of the execution, the results were automatically pushed to the Git repository, utilizing it for comparing the modified files.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Riccardo Sisto, for his availability and for all the advice he has provided me during this journey. I also want to express my gratitude to Ivan Aimale, Marco Oria, and Luisa Gatto for proposing this thesis project to me and for guiding me through this incredibly important path of personal and academic growth. Of course, I must also extend my thanks to all the colleagues with whom I have collaborated, as they have played a crucial role in my personal growth. I am proud to have been able to apply what I have learned at the Politecnico di Torino in a reputable company with highly skilled and supportive individuals.

I will never cease to be grateful to my mother, my father, and my sister for always being there for me and providing support not only financially but, above all, emotionally. If I have reached this point, it is primarily thanks to them.

I also want to thank my girlfriend, who, throughout my journey at the Politecnico, albeit from a distance, has always been by my side and has given me the strength to move forward without ever turning away. I thank her for all the moments, both beautiful and challenging, because thanks to her, I have been able to grow and improve myself.

A massive thanks also goes to my friends, who have always been available to help me and capable of bringing a smile to my face even in the most difficult moments.

Table of Contents

List of Tables

List of Figures

List of Codes

Acronyms

DB

database

OWASP

Open Worldwide Application Security Project

SQL

Structured Query Language

SAST

Static Application Security Testing

DAST

Dynamic Application Security Testing

CWE

Common Weakness Enumeration

DOS

Denial Of Service

LLMs

large-scale language models

Chapter 1

Introduction

Web application are becoming more and more common, they are easy-to-use and almost everyone has a device capable of surfing the internet. Whenever talking about an online service we may think that the most significant aspect is the user interface. Unfortunately, there is another aspect that usually is overlooked and it is the one of cybersecurity. Indeed, if an application is vulnerable it is a real threat not only for the company providing the service, but also for the final user, data disclosures can lead to password being discovered and in the worst scenario also credit card number could be stolen. At this point, needless to say, that having a secure application is another key aspect. Usually,developers do not have enough time to keep in mind the basic rules of secure coding, this could lead to introduce vulnerabilities of various type in the source code. It would be convenient to have a software capable of automatic fixing all the vulnerabilities, this would help to save huge amount of time.

1.1 Objective

This thesis was made in collaboration with the company Liquid Reply, and the aim of this work is to develop a tool capable of automatically fixing vulnerabilities in the source code. To achieve this, it was first analyzed the state of the art, seeking to understand what the market offers, highlighting the pros and cons of each solution found. Subsequently, it was the moment of the development phase, and finally, it was conducted the validation part, which involves assessing the quality of the results obtained.

1.2 Structure of the thesis

The thesis is composed of the following chapters:

1. **Chapter 1**, this one, where the thesis is introduced;
2. **Chapter 2**, dedicated to the background of some vulnerabilities and tools used to find them (like SonarQube);
3. **Chapter 3**, study about the state of the art of the tools capable of fixing vulnerabilities;
4. **Chapter 4**, explanation about the decision taken to implement the tool;
5. **Chapter 5**, the development of the tool;
6. **Chapter 6**, describes the validation phase, showing reports and percentage of fixing;
7. **Chapter 7**, conclusion of the thesis and possible future works;

Chapter 2

Vulnerabilities and how to detect them

In this chapter we will analyse what is a vulnerability and how to detect and fix it. We will focus on SQL injection and on SonarQube that are two of the main aspects of this thesis work.

2.1 Cybersecurity key concept

Whenever developing a software we must keep in mind the importance of cybersecurity. Security in this context means to protect any computer systems. Here there are some keyword that will help us to understand:

- **assets** are the resources that we want to protect because of their value, they are precious. Assets are data, they could be credentials of users or credit cards number, or anything that must be protected in a system;
- **threats**, they are basically what can compromise the security of our assets, not all threats can be used to provoke real danger to our system. The one that can be used are called exploitable;
- **vulnerabilities**, this is the actual name of “exploitable threats”, it is very important to detect them in order to prevent them.
- **security control** are all the countermeasures actuated to prevent damage to our system;
- **security requirements** are the standards of security that we want to achieve and what is needed to implement such mechanisms;

- **security risk**, it is a value indicating how much a system is vulnerable. It is calculated using all the previous concepts.

2.2 Vulnerabilities

Now that we have introduced the most important keywords, we can start speaking about what actually is a vulnerability. As we have discovered, it is an exploitable threat, and it could have been introduced in our system in any of the phase of the development. It usually is in the form of a bug or a flaw. They are similar but not the same thing.
A **bug** is an error that prevents the normal execution of our code, they may be caused by many factors, programmers have forgotten something in the source code, some user inputs are not well handled and so on.
On the other hand, a **flaw** is again a threat and could lead to a vulnerability, but doesn't necessarily mean that the execution of our program will be interrupted, for example if a password is stored in clear text, it won't be a problem from the point of view of the logic, but of course it could lead to the disclosure of this password in case someone is able to read the content of our DB. As you can see in this case the problem is in the design of our application and for this reason it is harder to find and solve this type of problems, we have to change many components of our system in order to solve the issue.
As we can see, not all flaws are bugs, but all bugs are flaws. As said before whenever they are exploitable they could lead to different types of vulnerabilities. Talking about all the vulnerabilities would be almost impossible, for this reason we are reporting here the top ten published by OWASP:

Position	**Type**	**Explanation**
1	Broken Access Control	The access control is in charge of enforcing policies. It means that only authenticated users with certain privileges can modify certain type of data [1]
2	Cryptographic Failures	If working with sensitive data, it is fundamental to use strong crypto algorithm [2]
3	Injection	When inputs' data are not checked this could lead to an injection. Some of the most commons are SQL and command injection [3]

Table 2.1 – continued from previous page

Position	Type	Explanation
4	Insecure Design	It happens when users do not use secure design patters. Whenever wanting a secure application it must be built including secure libraries and methodologies [4]
5	Security Misconfiguration	Security is not well configured. e.g.permission improperly configured [5]
6	Vulnerable & Outdated Components	This could happen when we use outdated library or components that are no longer secure [6]
7	Identification & Authentication Failures	It happens when an attacker can bypass the authentication phase and impersonate another user. Usually it is because we permit weak password or we use weak credential recovery mechanisms [7]
8	Software & Data Integrity Failures	When integrity of software/data is not checked for example when downloading libraries from untrusted repositories [8]
9	Security Logging & Monitoring Failures	When logging is not well integrated, attackers may attack without being discovered and/or detected [9]
10	Server-Side Request Forgery (SSRF)	An application is vulnerable to SSRF when it fetches a remote URL without validating it [10]

Table 2.1: OWASP top 10 software vulnerabilities (2021) [11]

The ranking was made according to one critical aspects: the most common issues that companies are facing nowadays, in other words the most critical vulnerabilities.

2.3 Vulnerabilities and time

Before discussing SQL injection (the vulnerability addressed in this thesis), it is important to emphasize the significance of the time factor. This particularly affects the lifecycle of every vulnerability.
Various scenarios can occur, but they all begin with the unintentional introduction of

a new vulnerability in a product. This is also known as a **zero-day vulnerability**, which means that even the vendor is unaware of it. From this point onwards, the evolution of this vulnerability depends on when two main events occur:

1. the development team becomes aware of the vulnerability;
2. the vulnerability is discovered by someone outside the company.

Once the company becomes aware of the vulnerability, they must strive to resolve it as quickly as possible, as the risk of someone developing an exploit increases over time. Talking about the former event (1), it can occur in three different ways:

1. in the best-case scenario, the vendor discovers the vulnerability and immediately begins working on a patch;
2. if it is discovered by another team and communicated to the vendor, a **grace time** is usually given, which is a deadline by which the vulnerability must be fixed, or else it will be publicly disclosed to the world. This practice is also useful to incentivize the development of a patch that might otherwise be postponed. This scenario is known as **responsible disclosure**;
3. the vulnerability is publicly disclosed to the entire world without any grace time. This is, of course, the worst-case scenario and the most delicate one, it is called **full disclosure**;

Once the company is aware of the vulnerability, it will work to develop the patch as quickly as possible to reduce the risk percentage to nearly zero (though the risk can never truly be zero).

As noted, the third case is the worst, and it is evident that it is a race against time between those defending (the companies releasing products) and those attacking (hackers or external research teams). Therefore, the less time it takes to fix the vulnerability, the better. The period of time from when the vulnerability is introduced until the patch is released and installed is referred to as the **window of exposure**, which is, once again, a measure of time. It is essential to always minimize this as much as possible. This highlights the fact that having a tool capable of automatically fixing vulnerabilities could be very helpful and of course would speed up the entire fixing process, reducing risks.

2.4 SQL Injection

As evident from the table in section 2.1 (OWASP top 10: 2021), the category “injections” ranks third, indicating that they are relatively common vulnerabilities and, most importantly, have a significant impact when exploited. Therefore,

preventing and addressing them is of utmost importance. In this section, we will discuss the vulnerability that the tool is capable of fixing: SQL injections. First and foremost, it's important to clarify that SQL stands for Structured Query Language, which is a language used for querying databases.
SQL injections are vulnerabilities where the attacker exploits unsanitized and unvalidated input to add SQL code to a query defined by the programmer in order to alter its behavior [12]. Typically, this is done to execute SQL commands or to modify the behavior of the original query, such as bypassing password checks. It is not important the language used, we must face and handle with SQL injection in all the possible programming languages: Java, PHP, Javascript... .

First of all, we must say that in SQL a common way to add a comment is with this sequence of characters "–". Now we can try to analyze the following line of codes:

```
    String sql = "SELECT * FROM users WHERE username= '"+username + "
   ' AND password='" + password + "'";

    Connection connection = db.getConnection();
    Statement statement = connection.execute(sql);
```

List of Codes 2.1: Example of SQL Injection in Java

This is a simple query that checks if in the DB there is that specific username with that password (this example was made just for simplicity, it is seriously dangerous to store in clear text passwords in the DB). An attacker could send the following input:

- "admin' –" as username;
- whatever as password, for example "123";

In this case the query would assume this form:

```
SELECT * FROM users
WHERE username = 'admin' --'AND password = '123';
```

List of Codes 2.2: Result of inputs in SQL

We can clearly see that by adding that comment, the password check is bypassed. Doing this, an attacker is capable of logging in as admin (in case user "admin" exists) and the password won't be checked. There are other approaches similar to this one, one of them is using an always true clause, such as "`1 = 1`", in this case changing the inputs we could achieve something like this:

```
SELECT * FROM users
WHERE username = '' OR 1=1 --'AND password = '123';
```

List of Codes 2.3: Result of inputs in SQL

Again this will lead to be logged in without entering any password, because "`1 = 1`" is always true, so it will match all the users.

This is just one type of SQL injection attack and it is part of the **tautologies**, but there are many others such as the **Union Query** which intent is to retrieve other sensitive data from a query by using the Union command of SQL. Another one is the **Piggy Backend**, where the intent is DOS (Denial Of Service) or altering table states [13].

To prevent this type of attacks there are many approaches, we will only talk about the easiest that doesn't completely alter the structure of our code. Instead of the variable "`Statement statement`" we will use another type `PreparedStatement`. The code will become as follow:

```java
1     String sql = "SELECT * FROM users WHERE username= ? AND password= ?";
2
3     Connection connection = db.getConnection();
4     PreparedStatement statement = connection.prepareStatement(sql);
5
6     statement.setString(1, username);
7     statement.setString(2, password);
8
9     statement.execute();
```

List of Codes 2.4: Example of correct query in Java

As we can see we don't concatenate username and password directly inside the string "`sql`", we have used placeholders "?". After that we must set each "?" and this is the reason why we have added those two lines of code (6 and 7). At this point the `PreparedStatement` understands that everything that is inside "`username`" must be replaced with the first "?" and if it contains SQL commands those will be escaped, so even if there are commands and/or comment in "`username`" and/or "`password`", they will be escaped and so, they will not be executed.

2.5 Vulnerability detection

We have analyzed some vulnerabilities, focusing on SQL injections. Now, however, we will concentrate on how to determine if the code we need to study contains them. This type of activity is called "security assessment" (we will omit the analyses on Networked Systems). First, we need to distinguish between two approaches:

- **Static**: the application is never launched. Testing is done by examining what is happening in the code itself. This involves checks on both semantics and variable types. Additionally, theorem prover techniques are adopted, and the flow of variables, as well as their sources, is controlled. To be even more precise, symbolic execution is performed, where a model is generated based on the original code (this phase should not be confused with actual execution). Typically, in this type of analysis, white-box tests are performed, which means that you have full access to everything, and your knowledge of how the program functions is at 100%.

- **Dynamic**: In this approach, the code is executed, and vulnerabilities are sought through testing. This type of analysis is accompanied by black-box tests, where you simulate not knowing anything and attempt to attack the application. During this phase, tools such as decompilers and/or disassemblers come in handy.

Depending on whether the analysis is static or dynamic, the tools are given the following names: **SAST** (Static Application Security Testing) and **DAST** (Dynamic Application Security Testing).

SAST has the following advantages

- it can find more vulnerabilities;
- it can point to the exact location in the code;
- the type of vulnerability is reported;
- it can be used in every phase of the development.

On the other hand, here there are SAST cons:

- since the code is not actually executed, false positives can be introduced in this type of analysis;
- each tool used is specific to a certain programming language.;
- it is also challenging to obtain good results regarding external libraries.

Talking about DAST, it has has some pros:

- introducing fewer false positives;
- it is not depending on the programming language used.

However, the cons are:

- it finds fewer vulnerabilities;
- it can only be used in the latest stage of development;

- it does not provide information about where the vulnerability was introduced in the code and what type it is.

Recently, a new type of analysis called **IAST** (Interactive Application Security Testing) has been introduced. It seeks to combine the advantages of both approaches to be more comprehensive like SAST while trying to reduce false positives like DAST. Its operation is similar to DAST, but it also includes context information in the code of the vulnerability[14].

2.6 SonarQube

In this section we will analyse SonarQube, a SAST, that is the one used in this thesis work to find vulnerabilities in the code. Talking about Sonar, it is an automatic code review tool that can be integrated into the workflow of any project. It is capable of detecting issues in the source code and providing info about these (bug, vulnerability, secure hotspot...). It is powerful because it works on several programming languages. It also provides extension for IDE to help the programmer since the beginning of the development [15].

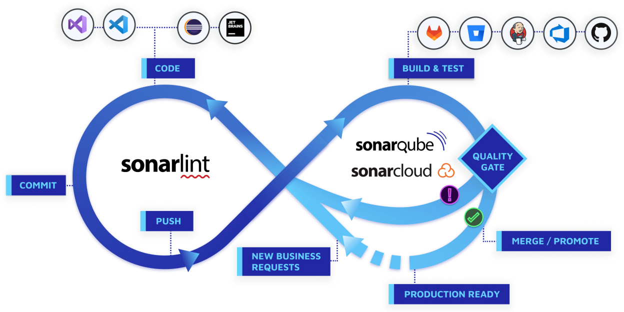


Figure 2.1: Sonar workflow [15].

You can select different quality profiles that will help you identifying each issue in your code, this is done because each profile contains a set of rules to be applied during code analysis. SonarQube comes out with the BUILT-IN profile, however the final user can define his own set [16]. The most powerful feature of SonarQube is the capability to integrate other plugins in a very simple way, indeed there is a section called "marketplace" that works like an app store. Among the countless available plugins, we will focus on FindBugs (or FindSecBugs), which offers four different quality profiles for Java. Their strength lies in the fact that among the various rules, they have specific ones for identifying vulnerabilities in the code. FindSecBugs has `141` bug pattern and it can be integrated in IDE, or can run locally, or (in our case) can be integrated with SonarQube. The tool has been developed keeping in mind the top 10 published by OWASP (OWASP top 10: 2021) and also CWE (Common Weakness Enumeration)[17]. Now we will focus on a specific rule, the one of SQL Injection, it it called "**Potential SQL Injection**", it refers to standard Java (Java vanilla) and it is different from the other rules that has a similar name (starting with "Potential SQL Injection") because those rules refer to other framework such as Hibernate or JPA and so on. As we can read from their website: "The input values included in SQL queries need to be passed in safely. Bind variables in prepared statements can be used to easily mitigate the risk of SQL injection"[18]. We are focusing on this rule because this is the one used by our tool to distinguish which vulnerabilities it is able to fix.

Chapter 3

State of art

In this chapter it will be discuss the state of the art of tools that are capable of automatically fixing vulnerabilities. This was the first step to understand what the market offers, after this chapter we will discuss the various solutions considering their pros and cons. We will start from SAST, and then we will move on to the latest solutions that are based on Artificial Intelligence: **ChatGPT**.

3.1 SAST tool, automatic fixing

As we said before SAST are tools capable of spotting vulnerabilities inside the source code. However, most of them are not capable of solving the issue. For example SonarQube can find some problem inside the code describing the vulnerability and possible attacks, but it cannot fix the code automatically. Anyway, developers are recently attempting to integrate features into their products to automatically correct some vulnerabilities. Considering that coding, despite the existence of standards, is a highly creative process, this implies that every program is different from another, even if they perform the same task. It is easily evident that fixing every vulnerability in every type of code is almost impossible, as there will always be cases that cannot be resolved without human intervention. Now, it will be reported some of the best solutions according to Gartner [19]. As we can read from their page, the market is evolving day by day, so everyday the ranking of a product could change.

3.1.1 Mend.io (formerly WhiteSource)

Mend.io is a company that offers various services, one of the most intriguing is Mend SAST because in their page we can read about its functionalities and it is said to be capable of fixing some vulnerabilities and there is also a picture showing it [20]. Furthermore, there is a demonstrative video at this link: `https://www.mend.io/mend-platform/?wvideo=zmkarqybre`. Standing to what others say:

- "Mend (formerly WhiteSource) does a great job of that and we had quite a few when we first put this in place. Mend does a very good job of finding the **open-source**, checking the versions, and making sure they're secure" [21];

- "The remediation-centric approach (automated remediation being the automatically created pull requests with the appropriate lines of code to be changed to resolve the issue) is a game-changing innovation." [22];

- "Mend has a security research team to find and evaluate vulnerablities faster than the entries in the NVD database" [23];

In an email we asked for having more information regarding how the tool operate, they only told us that the product is capable of doing all the stuffs that was written in their site. Basically it is able to fix the source code, and that they have started working keeping in mind the top ten produced by OWASP (OWASP top 10: 2021). This is a great step forward because all the other softwares usually can sanitize only the dependencies. Continuing the research on this company, it was discovered that the tool was not always able to fix vulnerabilities and still limited itself to a few simple cases.

3.1.2 Veracode fix

At the time of writing, Veracode is considered the best solution according to Gartner ranking [19]. It is a solution that offers both SAST and DAST. In June 2023, the development team decided to integrate a new feature called "**Veracode Fix**", which, leveraging machine learning, will become capable of automatically fixing code vulnerabilities. This will be based on artificial intelligence, similar to the one of ChatGPT, but of course, it also requires a good training dataset [24]. The suggested fixes come from the real word, and the team continuously updates this set. Again we have a video showing a demo: `https://share.vidyard.com/watch/gYTrj7MW1uUtrYqCBPsURR?`.
The problem is that this features came out in June and our deadline to decide what to do next was the beginning of May, at that time Veracode had released a demo of this functionalities, but it was not so powerful. It was not able to fix a SQL injection for example. As other tool at that time, it was able to fix vulnerabilities in the dependencies.

3.1.3 CheckMarx SAST

CheckMarx SAST is the second one in Gartner ranking [19], we discovered that actually it is not capable of fixing automatically. Its points of strength are absolutely providing a great integration with GitLab and GitHub (in the DevSecOps workflow) and also sorting issue according to their criticality [25].

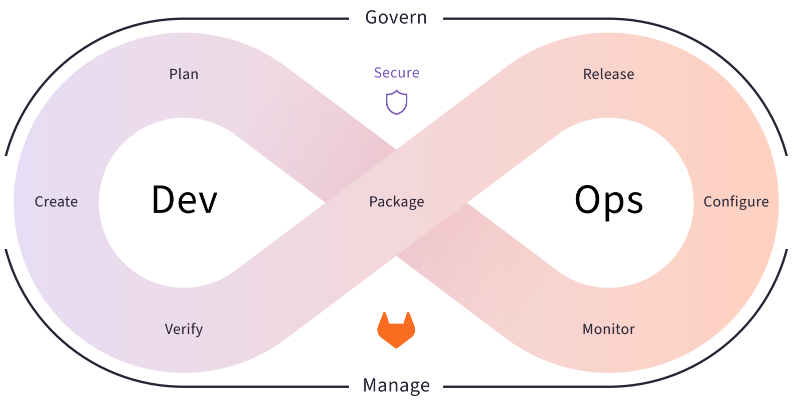


Figure 3.1: GitLab DevSecOps workflow [26]

Here there is a video demonstration on how CheckMarx SAST works `https://www.youtube.com/watch?v=pNlxHO7iRZY`.

3.1.4 Rapid7 InsideAppSec

Differently from the other this is a DAST tool, it is integrated in DevSecOps workflow like the others. It is said to be capable of recognising more than 95 different vulnerabilities. Its peculiarity is that is capable of making "**Attacks replay**", it means that directly from the generated report anyone can recreate the attack to exploit that specific issue. This makes easier the fixing procedure. Despite all those useful features, it does not fix automatically.

3.1.5 Snyk Code and Sourcegraph

We will discuss Snyk Code [27] and Sourcegraph [28] together because they have one common feature: fixing automatically dependencies. Often, programmers use existing libraries to assist themselves and avoid rewriting everything from scratch. However, in general, not all introduced dependencies are necessarily secure. In fact, if there is a vulnerability in the library's code, it could have repercussions on the application that uses it. To address this type of vulnerability, we must be careful about how libraries are used (if we are lucky the vulnerable part of the code may not be used), and unfortunately, in some cases, there is no choice but to wait for a patch to be released.

These two tools do just that: if they find dangerous dependencies for which an update is available, they perform it. Typically, it is sufficient to change the version number, for example, from "`version 1.2.1`" to "`version 1.2.2`". Clearly, doing this kind of work on huge amounts of libraries could be time-consuming, so having tools that automate this process is convenient, but unfortunately, it is not what we are interested in for the purposes of this thesis.

3.2 ChatGPT

3.2.1 ChatGPT and coding

With the advent of AI, the way we program has been revolutionized. There are now products like ChatGPT by OpenAI, which can interpret, generate, and/or correct the code we give as input. Recently, large-scale language models (LLMs) have also been developed, trained on massive datasets containing both human conversations and numerous examples of code. This has made AI increasingly proficient over time in communicating with us and even working on source code [29]. From this, projects like **GitHub Copilot** have emerged, capable of directly assisting us in the programming phase. The more code we write, the more precise this AI becomes and it can "emulate" our style.
We have tried ChatGPT to see how capable it is in fixing vulnerabilities. Only giving to it the vulnerable code, without adding anything else, the AI detects the vulnerability and fix it accordingly to what the user has asked. The response unfortunately was not always the same, so sometimes it was the fixed code, while other times it was the definition of the vulnerability found in the code. The former could be very helpful, while the latter could lead to serious problem if integrated in a tool that must correct the source code.

Another test was done on some common vulnerabilities present in a git repository [30], ChatGPT-3 was capable of detecting `213` vulnerabilities, much more than Snyk that **only** found `99` vulnerabilities. Both tools found many false positive, but anyway this experiment let us understand how powerful is this AI [31]. Just to let the user understand, we are reporting an example in Java:

```java
class Utils
{
    // Function to serialize an object and write it to a file
    public static void SerializeToFile(Object obj, String filename)
    {
        try
        {
            FileOutputStream file = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(file);

            // Serialization of the object to file
            System.out.println("Serializing " + obj.toString() + " to
    " + filename);
            out.writeObject(obj);

            out.close();
            file.close();
        }
        catch(Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
    }

    // Function to deserialize an object from a file
    public static Object DeserializeFromFile(String filename)
    {
        Object obj = new Object();

        try
        {
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);

            // Deserialization of the object to file
            System.out.println("Deserializing from " + filename);
            obj = in.readObject();

            in.close();
            file.close();
        }
        catch(Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }

        return obj;
    }
}
```

List of Codes 3.1: Java test on ChatGPT-3

Given that code, ChatGPT generated the follwing response:

Vulnerabilities Detected:

1. Insecure Deserialization: Deserializing untrusted data can lead to remote code execution.
2. Insecure File Permissions: The file being written to or read from may have insecure permissions, allowing unauthorized access.

This example is taken from the test described above [31]. It is crucial to highlight this test was made using ChatGPT-3, nowadays there is ChatGPT-4 that is even more sophisticated.

3.2.2 ChatGPT IDE integrations

To help developers, many plugins were introduced for the IDE, among them we are reporting the most significant one:

- **CodeGPT** is an extension for IntelliJ that allows you to send selected code directly from the IDE to ChatGPT, which can then correct it in case you are interested in fixing bugs or generate tests [32].
- As for Visual Studio Code, there is **GPT coder**, which, however, is not capable of having ChatGPT evaluate the code but can interact with it to generate code based on a description provided by the user [33].

Unfortunately, none of this tool is capable of automatically fixing the source code. Indeed, the former asks to ChatGPT to optimize the code and the response will contain the fix made by the AI but this is done randomly. It means that in some cases it is a non-vulnerable version, but in many other cases it is an optimization related to performance, that has nothing to do with the security of the source code (this is a problem of all the AI, it will be explained better in the next chapter). The latter is not capable of fixing the code, it can "only" generate new methods.

Chapter 4

Taken choices

As it can be seen from the previous chapter, none of the solutions available on the market is truly capable of automating the fixing process. For this reason, it has been decided to develop a tool from scratch. To do this, there are two viable paths:

- AI (Artificial Intelligence), ChatGPT;
- standard code;

In this chapter, both of these possibilities will be explained, highlighting their pros and cons.

4.1 ChatGPT integration

After analyzing the capabilities of OpenAI's AI, it was decided to conduct tests to understand how it performed with various examples. In this case, there are two possible options:

1. Open the browser and send the request through chat.
2. Use the APIs.

Using the first solution has several limitations, as the user does not have freedom to choose the AI's behavior. The second option allows for a higher level of configuration. In both cases, being a language model, the response can vary even if the question remains the same.

4.1.1 ChatGPT from the browser

The browser approach is straightforward, requiring only an account to query the AI. ChatGPT (version 3.5) is capable of generating solutions for various vulnerabilities in many programming languages, although it does so with limitations. For instance, if the problem is already known, you can send the code and ask it to fix that particular vulnerability. The response is generally quick, although it may occasionally get blocked due to excessive requests. However, when developing a tool, it is much simpler to invoke the APIs.

4.1.2 OpenAI API

This solution provides users with various levels of customization. You can choose the **model** and **version** that should generate the response, adjust various parameters that control the AI's freedom during generation (e.g., **temperature**), and instruct the model on how to behave when generating a response (e.g., generate the response as if you were an experienced programmer). All these factors help tailor requests and make them more precise. Moreover, adjusting the parameters also means trying to limit the variation in responses.

4.1.3 Results

The results obtained show that the second approach is the better one. In both cases, SQL Injection, XSS, Command Injection, and Path Traversal vulnerabilities were corrected in many programming languages, such as JavaScript, Java, PHP, and more. To test the APIs, a **Python script** was written to connect to the APIs using an auth-token. The AI was asked to fix SQL Injection with the question "Can you fix the SQL injection in this code?" The response was then analyzed with various degrees of `temperature`. It was observed that reducing this parameter made the responses much more deterministic. However, it often resulted in incorrect responses. So, reducing the AI's freedom too much led to consistently identical responses to the question. In seeking a middle ground, it was noticed that unfortunately, the response shifted from being static and incorrect directly to a dynamic version. Therefore, the middle ground was not achievable. It was also observed that the APIs were quite slow. It took 3 hours to perform 100 tests because each request took at least 30 seconds and often failed, requiring a retry. However, the results were quite positive, as out of 100 tests, the AI only responded incorrectly 8 times (incorrect code or a response containing the vulnerability's definition).

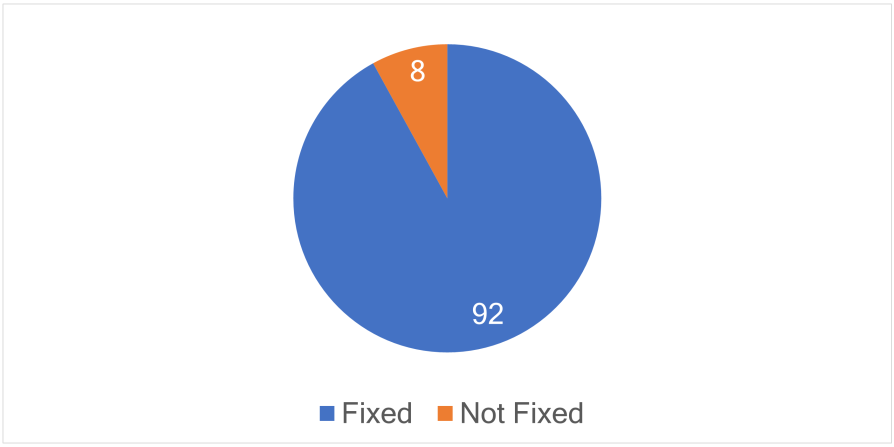


Figure 4.1: ChatGPT results.

Once the testing phase was completed, the pros and cons of the APIs were studied.
Pros:

- Corrects many vulnerabilities.
- Corrects in many different programming languages.
- Improves with updates from OpenAI.

Cons:

- Often fails and requires repeating the question in response.
- Takes approximately 30 seconds for each response.
- Requires an account to invoke the APIs, and there is a small cost for each token sent and received.
- Code to be submitted needs to be selected for privacy reasons (loses the great potential of ChatGPT to fix various vulnerabilities using existing libraries).
- Token limit for each question.
- The response changes even if the question remains the same.
- Being tied to OpenAI, it's uncertain how the company will behave with the APIs (changes, removal).

4.2 Standard code

Given all the various flaws of AI, it was decided to abandon that path and focus on **standard code**. Developing a new application from scratch solves many of the problems seen before, such as latency, privacy concerns, and also because the effort required to select pieces of code to send to ChatGPT (again, for privacy reasons) or to directly solve the problem with a new tool is equivalent. However, there are also drawbacks to this approach. In fact, AI is capable of understanding and resolving almost any vulnerability, while writing code capable of performing this type of processing would take too much time. For this reason, some choices have been made to make this thesis feasible.

4.2.1 Choosen vulnerability

As for the vulnerability, **SQL Injection** was chosen, which still ranks third in the OWASP ranking (`Chapter 2.2`). In fact, if present in the code, it could be used for attacks with multiple purposes. In the best case, it could lead to a **Denial of Service** (DOS) attack, but in the worst cases, it could also be exploited to obtain users' private data. Clearly, it was necessary to focus on a single programming language, and for this reason, **Java** was chosen. It is one of the most widely used languages in web development. It was decided to address only the **vanilla** version of this vulnerability, i.e., the case where parameters are concatenated into a statement that is then passed to the database, without the use of plugins (SpringBoot, Hibernate, JdbcTemplate, etc.).

4.2.2 Choosen language

Once the vulnerability was chosen, it was also necessary to decide in which language to write the tool. To keep this phase from becoming too resource-intensive, an object-oriented language was needed. Since there was no need to work with network connections (for which Python would have been suitable), Java was chosen. This language has many libraries that can assist during programming, and since the goal is to fix SQL Injection in **Java**, it makes it easier to attempt code parsing.

Chapter 5

Development of the tool

As is evident from the previous chapter, it was decided to develop the tool from scratch in Java so that it could automatically correct SQL Injections in Java code. Before proceeding with the actual development, it is important to discuss the structure of the following chapter. Firstly, the idea will be described, then the various libraries used will be introduced, and finally the actual code will be presented. Talking of the last one, it will be analysed the main logic and and the integration with git.

5.1 The idea

To address this issue, the first step was to study the type of vulnerability and attempt to develop an algorithm by outlining the necessary steps to tackle the entire process. It was assumed that:

- the input would consist of a report (from SonarQube) containing various code vulnerabilities.
- the output would be shown using Git.

Analyzing how to fix a SQL Injection (Chapter 2), it was evident that certain steps were the same each time, so it was possible to design an algorithm. However, there was another criticality: Java had to be able to understand other Java code because before fixing each vulnerability, it's necessary to comprehend the problem at the code level. Therefore, it was essential for Java to be able to parse other Java files (this will be addressed in the next section). Once this problem was solved, we could begin to consider the various steps to be executed in summary:

1. Read the vulnerable line.
2. Change the type to `PreparedStatement` (if it's not already).

3. Recursively search for all parameters used in the SQL query, simultaneously saving those that need to be set in the final part. This mean that starting from the vulnerable line a new replacement, consisting of a fixed line, is saved. Then all the parameters involved in the query are studied and for each line where they are involved is produced a replacement's line (if needed). Each parameter in the new replacement is searched in this way recursively (saving new fixed line). Whenever a new variable that must be set later is found, it is saved.

4. Replace all vulnerable lines with revised versions. All the replacements saved at point 3 are applied.

5. Add the setting part: `stmt.setString(1, username)`.

6. Add the `execute` line.

As it can be seen from the above list, some clarifications need to be made. The first one is about the type of Statement present in the code:

- Completely vulnerable query, thus having a Statement that needs to be changed to `PreparedStatement`.

- Partially vulnerable query, with a poorly written `PreparedStatement`, so it needs to be refined, and the order of variables in the setting phase must not be altered.

Other cases to be handled are all the forms in which each vulnerable line may appear. It could be a variable declaration or a simple assignment. In addition, the result of the execution is sometimes not useful (“`INSERT`”) and other times essential (“`SELECT`”).

Furthermore, it's necessary to consider that variables must be studied; it must be determined what comes from the user and what is already present in the code. It's also important to pay attention to various scopes; a variable may be used in multiple places, and when modifying it, it's crucial that it can still be visible in subsequent usages. Moving the declaration of a variable inside an “if” statement will make it no longer visible outside.

```java
public void example(int a){
    String result = new String();

    if(a%2==0){
        result = "EVEN";
    }else{
        result = "ODD";
    }
    System.out.println("Number is: " + result);
}
```

List of Codes 5.1: Java example of variable used in different scopes

In this case defining "`result`" at the beginning is fundamental, if the code was like this, then there would have been a compilation error:

```
public void example(int a){
    if(a%2==0){
        String result = "EVEN";
    }else{
        result = "ODD";
    }
    System.out.println("Number is: " + result);
}
```

List of Codes 5.2: Java wrong example of variable used in different scopes

This because "`result`" is not defined both for the `else` clause and for the `body` of the method. the only place where we can use this variable is inside the `if`. All these cases significantly complicate the algorithm above but must be managed to avoid altering the program's functionality and the behavior of a query.

5.2 Dependencies

In this section we will discuss the dependencies used in the code, some of them have a key role, because they are involved in the core logic, others are used only for helping during input/output.

5.2.1 JavaParser

JavaParser [34] is the library that addresses the first major challenge of the entire project: enabling Java to parse Java code. We will now delve into explaining how this library works because it plays a truly pivotal role; in fact, the entire project revolves around it. Let's begin by stating that it is a project based on the work of Sreenivasa Viswanadha and Júlio Vilmar Gesser [35], which commenced in 2008. Since then, this library has been consistently updated to align with various Java updates.

Using this library, it is possible to provide a Java file, and then it will be parsed. The obtained result is a **`CompilationUnit`** [36], the Java file in its parsed version, i.e., essentially, its syntax tree. Therefore, there is a section dedicated to imports (where other can be added). It is possible to search for one or more pieces of code and then operate on them using streams.

Before proceeding, let's also introduce the concept of **`Node`**. In this JavaParser, everything inherits from this class. Every variable, code block, method, and

`CompilationUnit` can be considered a `Node`. This type provides several useful methods, such as "`findAll()`" and "`findFirst()`", where you can specify what you want to find within that node. There are methods like "`getParentNode()`" to get the node before the one under analysis. For example, if the Node is an "`if`" statement within a method, "`getParentNode()`" would return the method's body. Then there are also methods like "`isAncestorOf()`" to determine if one node is the parent of another one. Another valuable method is "`replace(Node)`", which allows replacing nodes. The `Node` also contains information about the **`Range`**, basically the starting/ending lines and columns.

In addition to the Node, we also have the **`MethodDeclaration`**, which represents the declaration of a method. It includes the method's arguments, return type, and the method's body, which is a **`BlockStmt`**.

To understand what it is, it is needed to first define what a **`Statement`** [40] is. It's a class that extends `Node` and can contain one or more lines of code. Statements are generic, but from this class, several specific types inherit:

- **`BlockStmt`**: One or more lines of code enclosed in curly braces {...}. Inside, there are other Statements, so multiple BlockStmts can be nested. It has very useful methods that allow you to obtain a NodeList, which is a list of Nodes on which you can perform operations as if it were a regular list. The most important method is "getStatements()", which allows you to get this NodeList [41].
- **`IfStmt`**: It's a `BlockStmt` but also contains information about the condition and any "else if" or "else" clauses present afterward [42].
- **`WhileStmt`**: This is also a `BlockStmt` with loop conditions [43].
- **`DoStmt`**: Similar to `WhileStmt` [44].
- **`ForStmt`**: It also includes various conditions and a `BlockStmt` [45].
- **`TryStmt`**: It's the `Statement` for a try block [46].
- **`ExpressionStmt`**: These are Statements that contain an `Expression` within them, usually a single line of code [47].

To clarify, let's now discuss **`Expressions`** [48], which are the "units" of code. Like `Statements`, `Expressions` are extended by other classes:

- **`AssignExpr`**: It represents simple assignments like "`a = 5`" or "`a = b+1`". They are characterized by the operator used, which can be various, such as "+=", "=", "-=", "++", and also by the variables involved, which are `NameExpr` [49].

- **`NameExpr`**: It's an expression that represents a variable, in all instances except during declaration [50].
- **`VariableDeclarationExpr`**: This is the Expression that identifies the declaration of new variables, like "int a = 5" or "String s = new String()". Inside, they have a VariableDelcarator, which is how the declaration is defined. During the definition, the variables are not considered NameExpr [51].
- **`MethodCallExpr`**: It represents the invocation of a method, like "`a.getName()`". It can have multiple components: **arguments**, which are `NodeList<Expression>` (a list of Expressions), representing the parameters passed to the function; **scope**, which can be present or absent (it's optional), referring to the variable to which a method refers. For example, "`a.getName()`" has "a" as its scope, while "`getVar()`" has no scope. Finally, there's the method **name**, which can be obtained as a String or a `SimpleName`, that is another type for defining names [52].
- **`ConditionalExpr`**: It represents a condition in the code using the "?" operator, i.e. `String a = b.equals("Hello") ? b : "World!"`. It is composed of three Expressions: `condition`, `then` and `else` [53].
- **`StringLiteralExpr`**: It is a String in the code, i.e. String sql = "SELECT * FROM table", the String "SELECT * FROM table" is a `StringLiteralExpr` [54].

In general, both `Expressions` and `Statements` have methods for casting them to other types and `boolean` functions to determine if a certain Expression or Statement is of a particular type. For example, "`isAssignExpr()`" or "`isForStmt()`".

Now that we have analyzed the components of this library, let's discuss the **`TypeSolver`** [55]. It's the component that allows to understand variable types. In the tool, it was used a **`CombinedTypeSolver`** [56], and below is its configuration.

```
public class Modules {
  public static String projectPath;
  public static String src;
  public static JavaParserTypeSolver typeSolver;
  private JavaSymbolSolver symbolSolver;

  public Modules(String path, String javaFile, List<String>
   dependencies) throws IOException {
    projectPath = path;
    src = projectPath+javaFile;

        /*Some lines where removed because not useful to understand
   the type solver*/

```

```
        typeSolver = new JavaParserTypeSolver(new File(src));
    CombinedTypeSolver combinedTypeSolver = new CombinedTypeSolver();
    combinedTypeSolver.add(new ReflectionTypeSolver());
    combinedTypeSolver.add(typeSolver);
    for(String dependecy : dependencies) {
      combinedTypeSolver.add(new JarTypeSolver(dependecy.strip()));
    }
    ParserConfiguration parserConfiguration = new ParserConfiguration
    ();
    this.symbolSolver = new JavaSymbolSolver(combinedTypeSolver);
    parserConfiguration.setSymbolResolver(this.symbolSolver);
    javaParser= new JavaParser(parserConfiguration);
  }
}
```

List of Codes 5.3: Configuration of TypeSolver

As we can see, to be more precise the one in charghe of understanding the types is the **`JavaSymbolSovler`** [57], that accept as parameters the `TypeSolver`. In this code we have:

- `src` is the directory containing the various ".java" files. These files are needed to understand if the type in question has been defined by the user. To do this, you must grant JavaParser access to all the source files;
- `ReflectionTypeSolver` is a type that instructs `JavaParser` to search for various types among the passed files [58];
- `dependencies` are all the Jar files required for a program to function (external libraries). All of these must be provided; otherwise, `JavaParser` will struggle to interpret types from the libraries, such as `HttpRequest`, for example;

Another peculiarity of JavaParser is the **`Visitor`** class, it can be configured to find whatever we want in the code, here there is an example:

```
    private class MethodVisitor extends VoidVisitorAdapter<Map<String
    , Method>> {
    private Module mod;

    public MethodVisitor(Module m) {
      this.mod = m;
    }
    @Override
    public void visit(MethodDeclaration m, Map<String, Method> res) {
      res.put(m.getNameAsString(), new Method(m, this.mod));
    }
  }
```

List of Codes 5.4: Example of `Visitor`

In this code we can see that, starting from a **`Module`**, that in the tool is a class that contains a `CompilationUnit`, we can extract all the `MethodDeclaration` inside this `CU`, and save them inside a `Map` (this has as key the name of the method and as value a **`Method`**, a class created in order to store the `MethodDeclaration` and also other information).

5.2.2 StreamEx

`StreamEx` [59] is a library that addresses the limitations encountered in classic Java `Stream`. The main reason for using it was the ability to define additional conditions in the "`distinct()`" operation. Normally, as also stated in the documentation [60], to perform comparisons between elements, the "`equals()`" method of the object under analysis is invoked. In JavaParser, when "`equals()`" is invoked, it compares the type first, so whether it's a `Statement`, `Expression`, or anything else, and then the content. In our case, this allows to take two different pieces of code and compare them, when the two pieces of codes contain the same code, they would be considered equals (independently from the position in the code). However, in our case, there are times when we need to ensure that a piece of code is not repeated multiple times, so we need to compare the Range as well (i.e., the position in the code). To solve this problem, there are multiple approaches. The first is to modify the equals method defined in the various classes, but then it is lost the ability to compare pieces of code that are different, meaning that if two `NameExpr` instances (i.e., two variables), that don't come from exactly the same code location, are compared the result would be "`Not equals`". However, this feature is very useful, especially for identifying identical variables in different parts of the code. The second solution (that is also the one adopted) is to use StreamEx, so that it can be specified what to perform the comparison on, in this way you do not need to modify the "`equals()`" method. Here there is an example:

```
static List<Part> splitString(Expression expr, Method method){
    return StreamEx.of(expr.findAll(Expression.class))
    .filter(e-> e.isStringLiteralExpr() || e.isNameExpr())
    .map(e -> getBiggerExpressionUntilOrigin(expr, e))
    .distinct(e -> Arrays.asList(e, e.getBegin().get(), e.getEnd().
   get()))
    .map(e -> new Part(expr, e, method))
    .collect(Collectors.toList());
```

List of Codes 5.5: StreamEx example of distinct()

This function starting from an `Expression`, finds all the inner Expressions, then it creates a Stream (using StreamEx) and then operates on the results. Firsty we check rather it is a `StringLiteralExpr` or a `NameExpr`. After this filtering operation, we get the biggest Expression, before the one passed to the function, in this case before `expr`. Now we want to remove the duplicated results. For example in case both the `then` and the `else` clause of a `ConditionalExpr` are a `StringLiteralExpr`, then getting the bigger Expression would lead to have the same `ConditionalExpr` twice. This is the reason why, a `distinct()` is needed, now we will explain the reason behind the usage of the `Begin` and the `End`. In the same Expression we may have variable repeated, if those are used inside the same Expression the distinct will cut out one of them, using the position we will additionally compare this information in order to know if they are actually the same `Expression`. The rest of the code will be explained in the section dedicated to the logic of the tool.

5.2.3 json

This library is not essential; it is used for input and output purposes, can read and manipulate objects in `JSON` format, and can convert them to `CSV` [61]. It can also perform the reverse operation. Since the report extracted from SonarQube is in JSON, it is very convenient to use this package. It is also useful to print out the `CSV` containing the result.

```java
public void readCSV() throws IOException {
     String csv = new String(Files.readAllBytes(Paths.get(this.
    vulnerabilityCSV)));
     JSONArray json = CDL.toJSONArray(csv);

          ...

  }
```

List of Codes 5.6: Example of usage json

5.2.4 Maven Invoker Plugin

Maven Invoker Plugin [62] is also a library primarily used for input-related reasons. As can be seen from the section regarding `TypeSolver`, all the `JAR` files are required to make the application work, including those related to dependencies. Through this library, it is possible to launch Maven commands directly from Java. Below is the code to obtain the list of complete paths for each `JAR` of the dependencies. This process is carried out for every “`pom.xml`” file found in the project.

```java
private String[] mavenDependencies(String builder) throws
    MavenInvocationException {
     String mavenHome = System.getenv("MAVEN_HOME");
```

```
 3    if(mavenHome == null) throw new IllegalStateException("To use maven you must specify an ENV variable that point to the Maven directory");
 4    InvocationRequest request = new DefaultInvocationRequest();
 5    request.setPomFile(new File(builder));
 6
 7    request.setGoals(Collections.singletonList("dependency:build-classpath"));
 8
 9    Invoker invoker = new DefaultInvoker();
10    invoker.setMavenHome(new File(mavenHome));
11
12    ByteArrayOutputStream output = new ByteArrayOutputStream();
13    PrintStream printStream = new PrintStream(output);
14
15    InvocationOutputHandler outputHandler = new PrintStreamHandler(printStream, true);
16    InvocationOutputHandler errorHandler = new PrintStreamHandler(printStream, true);
17
18    request.setInputStream(InputStream.nullInputStream());
19    request.setOutputHandler(outputHandler);
20
21    InvocationResult res = invoker.execute(request);
22
23    List<String> lines = Arrays.asList(output.toString().split("\n"));
24    int indexDependecies = lines.indexOf(lines.stream().filter(l-> l.contains("Dependencies")).findFirst().get()) + 1;
25    return lines.get(indexDependecies).split(";");
26
27  }
```

List of Codes 5.7: Maven dependencies Jar

As we can read from line 7, the command executed is "`mvn dependency:build-classpath`"

5.2.5 Gradle Tooling API

Gradle Tooling API [63] is the equivalent of Maven in Gradle. In this case, the process is slightly more complex because before proceeding, we need to define a new task in the "`build.gradle`" file, specifically to generate the list of JAR files required to initiate and execute the process.

```
private List<String> gradleDependencies(String builder) throws IOException {

    GradleConnector connector = GradleConnector.newConnector();
    String gradleHome = System.getenv("GRADLE_HOME");
    if(gradleHome == null) throw new IllegalStateException("To use gradle you must specify an GRADLE_HOME that point to the Gradle installation directory");
    connector = connector.useGradleUserHomeDir(new File(gradleHome));
    File build = new File(builder);
    connector.forProjectDirectory(build.getParentFile());
    ProjectConnection connection = connector.connect();

    GradleProject project = connection.getModel(GradleProject.class);

    if(project.getTasks().stream().noneMatch(t-> t.getName().equals("printDependencies_VulnerabilityFixer"))) {
      FileWriter writer = new FileWriter(build, true);
      String taskToAdd = """
        task printDependencies_VulnerabilityFixer {\r
        doLast{\r
        configurations.runtimeClasspath.each { println it }\r
        }\r
        }
        """;
      writer.write(taskToAdd);
      writer.close();
    }

        ByteArrayOutputStream output = new ByteArrayOutputStream();
        try {
            // Create a launcher for the 'clean' task
            BuildLauncher launcher = connection.newBuild();
            launcher.forTasks("printDependencies");

            launcher = launcher.setStandardOutput(output);

            // Run the task
            launcher.run();
        } finally {
            // Close the connection
            connection.close();
        }

    return Arrays.asList(output.toString().split("\r\n")).stream()
        .filter(d-> d.endsWith(".jar"))
        .toList();
  }
```

List of Codes 5.8: Gradle dependencies Jar

The task to add is the one inside the variable `taskToAdd`, firstly we check if this task, called **`printDependencies_VulnerabilityFixer`**, already is in the file. If it is, then nothing is added, otherwise the new task is added to the "`buld.gradle`".

After that, we execute the task and we get the list of `Jars`. This process is repeated for each “`build.gradle`” file in the project.

5.2.6 Commons CLI

Commons CLI [64] is a necessary library for processing input parameters. It allows you to define the parameters that will be used and then save them. In the case of our tool, we have defined commands such as:

- “-d” or “-directory”, for the project directory, if not provided, it defaults to the current working directory.
- “-s” or “-source”, for specifying where the files are located in the project passed earlier, if not provided, it defaults to “src/main/java”.
- “-r” or “-report”, to specify the name of the file from which to take the initial report; if not defined, it defaults to “report.csv”.
- “-out” or “-output”, the name of the report printed after execution, if not present, it defaults to “Fixed_vulnerability.csv”.

Here there is the code to configure this parameters in the project:

```java
        Option input = new Option("d", "directory", true, "input directory");
    Option source = new Option("s", "source", true, "java library location");
    Option report = new Option("r", "report", true, "report file name");
    Option output = new Option("out", "output", true, "output name of the report");
    Options options = new Options();
    options.addOption(input);
    options.addOption(source);
    options.addOption(report);
    options.addOption(output);
    CommandLineParser cliParser = new DefaultParser();
    CommandLine cli = cliParser.parse(options, args);
    String projectPath = cli.getOptionValue("d");
    String javaFiles = cli.getOptionValue("s");
    String vulnerabilityCSV = cli.getOptionValue("r");
    String outputReport = cli.getOptionValue("out");

    if(projectPath != null) {
      this.projectPath = projectPath;
    }
    if (!(this.projectPath.endsWith("/") || this.projectPath.endsWith("\\"))){
      this.projectPath += "/";
    }
    if(javaFiles != null) {
      this.javaFiles = javaFiles;
    }

    if(vulnerabilityCSV != null) {
      this.vulnerabilityCSV = vulnerabilityCSV;
    }

    if(outputReport != null) {
      this.outputReport = outputReport;
      if(!outputReport.startsWith("\\") && !outputReport.startsWith("
  /")) {
        this.outputReport = "/"+ this.outputReport;
      }
    }
```

List of Codes 5.9: Setting of commons CLI

5.2.7 Jgit

Jgit [65] is a library that allows to make Git calls directly from Java code. It has been used for output purposes, and its functionality will be described later. For now, let's mention that to use it, it is sufficient to provide the username and password, or preferably an access token valid for git.

5.2.8 GitHub API

GitHub API [66] is an added dependency for output purposes. Indeed, JGit is unable to perform certain operations, so this library handles communication with GitHub. Again, username and password/token are required to function. In our case, this library manages "pull requests".

5.2.9 Http Client

To perform "merge requests" in GitLab, the equivalent of GitHub's "pull requests", we have chosen to directly query the APIs. Therefore, `HTTP Client` [67] is required to interact with GitLab's APIs. In this case as well, an access token is necessary.

5.3 Development, core logic

In this section, we will discuss the underlying logic of the program. As previously mentioned, for simplicity, we will assume the following:

- the SonarQube report is available (and preferably has been filtered for false positives);
- the project is already on Git and up-to-date (no commits are required);

That being said, we can proceed with the actual code description, keeping in mind that the mechanisms handling input and output will be explained later.

5.3.1 VulnerabilityFixer

The first class to be initialized is the **`VulnerabilityFixer`**, its contructor takes as input all the args given to main. At this point, as exposed in the previous section (`here`), the commons CLI package will be capable of setting the needed variables. After that piece of code, the modules variable will be instanciated. The `VulnerabilityFixer` type exposes three methods:

1. **`readCSV`**, the one in charge of loading the report produced by SonarQube, the list of vulnerabilities is also sorted according to the name of the file and then to the line where it appears.
2. **`fix`**, responsible of instanciating each class of the vulnerabilities and applying the fixing solutions.
3. **`exportResults`**, it is used to handle and print the result on Git and/or locally.

5.3.2 Modules, Module and Method

As the first step, the **`Modules`** variable is initialized, which receives the complete path to the project to be fixed along with the name of the source code folder. At this point, as can be seen from the schema mentioned above (this one), the `SymbolSolver` and `JavaParser` are configured, which will be used globally to interpret the Java code. This type is characterized by the following fields:

```java
public class Modules {
  private Map<String, Module> modules;
  public static String projectPath;
  private String src;
  private JavaParserTypeSolver typeSolver;
  public static JavaParser javaParser;
  private JavaSymbolSolver symbolSolver;
  private boolean allModulesLoaded = false;

 ...
 }
```

List of Codes 5.10: fields of Modules

Now we will analyse the variables not already discussed:

- **`modules`** is a map containing a Module, it is useful to get a module using its name (its relative path);
- **`allModulesLoaded`** is a boolean used to know if all the `Java` files have been loaded;

Before proceeding, it is needed to explain also the class **`Module`**. It is particularly useful because it is the one containing the `CompilationUnit`. We are reporting its initialization and its fields:

```
 1 public class Module {
 2 
 3   private CompilationUnit cu;
 4   private Map<String, Method> methods;
 5   private String name;
 6   private List<String> importsToAdd = new ArrayList<>();
 7   private Modules modules;
 8 
 9   public Module(String projectPath, String filePath, Modules modules)
      throws FileNotFoundException {
10 
11     this.modules = modules;
12     this.cu = Modules.javaParser.parse(new File(projectPath+filePath)
     ).getResult().get();
13     this.name =filePath;
14     this.methods = new HashMap<String, Method>();
15     new MethodVisitor(this).visit(this.cu, methods);
16   }
```

List of Codes 5.11: Module initialitation

This code is basically creating a new Module starting from a `Java` file (`filePath` variable), we are also saving a reference to all the Modules. Then we are creating a list with all the missing `imports` that are needed: `importsToAdd`. Finally we got two instructions at line 15 and 16. The former is just a definition of a new Map, that has as key a String and as value a class called Method. The former is the usage of what we have seen before while talking of the visitors, basically "`this code`" is extracting all the methods and is saving them inside this map. Now is time to discuss about the **`Method`** class. Here there is the definition of each field and also its constructor, we will summarize its usage and we will see the full details when we will talk about the algorithm.

```
1 public class Method {
2   private MethodDeclaration method;
3   private List<Parameter> arguments;
4   private String sequelIndex;
```

```java
  private List<ExpressionStmt> variableToAdd;
  private Map<VariableDeclarationExpr, List<ExpressionStmt>>
    variableDeclarationDependency;
  private Map<VariableDeclarationExpr, List<Entry<ExpressionStmt,
    NameExpr>>> addedDeclaration;
  private List<Statement> deleteList;
  private Map<ExpressionStmt, NameExpr> SQLNames;
  private Map<VariableDeclarationExpr, Set<Statement>>
    initDeclaration;
  private Module module;

  public Method(MethodDeclaration method, Module module) {
    this.method = method;
    this.arguments = this.method.getParameters();
    this.variableDeclarationDependency = new HashMap<>();
    this.addedDeclaration = new HashMap<>();
    this.variableToAdd = new ArrayList<>();
    this.deleteList = new ArrayList<>();
    this.SQLNames = new HashMap<>();
    this.initDeclaration = new HashMap<>();
    this.module = module;
  }

    ...

}
```

List of Codes 5.12: definition of Method class

Starting from the top we got:

1. `method` is the `MethodDeclaration`, so what this `Method` class is describing;
2. `arguments` are what the method is accepting;
3. `sequelIndex` is a String that identify the name to give to the index used inside the PreparedStatement during the setting phase. i.e., `stmt.setString(index, name)`, in this case "`index`" is the `sequelIndex`;
4. `variableToAdd` is a list of `ExpressionStmt` that contains all the new needed variable, they will be initialized at the beginning of the method. e.g., if `sequelIndex` is not defined, it will be added to this list.
5. `variableDeclarationDependency` is a Map containing as key a `VariableDeclarationExpr`, while as value it has a list of `ExpressionStmt`. This is done to trace all the usage of a variable. For example if a variable is used four times, then we will got its declaration as key and as value the four expession where this variable is used.

6. `addedDeclaration` is again a Map that contains all the information regarding a specific `VariableDeclarationExpr` and the various `ExpressionStmt`. As we will see, this is used to maintain the dependencies of the variables. So, if the original developer has used only one variable and, every time, this is riassaigned to avoid the declaration of a new one, then the tool will try to mantain this behaviour, but to do so, it is needed to know the dependencies of each declaration and their name (it will be analysed better).

7. `deleteList` is the list of the Statement that are no more needed and so they can be removed.

8. `SQLNames` is again a variable to track all the name usage. It is fundamental in case the tool have declared variables like this: "`stmt`", "`stmt_1`", "`stmt_2`", ... , "`stmt_n`". Whenever possible, it will try to replace each name with the one with the lowest value;

9. `initDeclaration`, in case of a initialization of some properties of the `Statement` (i.e., "`statement.setMaxRows(int row)`") the tool will save those and will put them in the correct place.

The constructor consists only in the creation of these Lists and Maps.

5.3.3 Vulnerability and SQLInjection

The tool was initially designed to correct all types of vulnerabilities. Unfortunately, there wasn't enough time to perform such extensive work. However, it was decided to keep an abstract class called **`Vulnerability`**, which was intended to serve as a wrapper for all other vulnerabilities. Currently, it is extended only by the **`SQLInjection`** class. The parameters accepted by this class are as follows:

- The **line** where the vulnerability is located.
- The **location**, which is the module in which it is found.
- The **modules** variable, which is a list of all modules, providing access to more general information and the ability to search within the entire codebase.

Once these parameters are accepted, the constructor of `Vulnerability` that accepts all three parameters is called directly. Below is the definition of this class and its corresponding constructor:

```
abstract class Vulnerability {

  protected int lineNumber;
  private String location;
```

```
 5   protected Method method;
 6   private Modules modules;
 7   protected Module module;
 8   protected Statement stmt;
 9   protected List<Variable> variables;
10
11   public Vulnerability(int line, String location, Modules modules)
       throws IOException {
12     this.lineNumber = line;
13     this.location = location;
14     this.modules = modules;
15     this.module = this.modules.getModuleByName(location);
16     this.method= this.module.getMethodFromLineNumber(this.lineNumber)
       ;
17     this.stmt = this.module.lineFinder(this.lineNumber, this.
       lineNumber);
18     List<Expression> expressions = JavaParserUtil.extractAllVariables
       (this.stmt);
19     this.variables = expressions.stream().map(Variable::new).collect(
       Collectors.toList());
20   }
21
22   public abstract void fix();
23 }
```

List of Codes 5.13: Vulnerability class

The input parameters are saved as protected so that the `SQLInjection` class can also access them. Afterward, the `getModuleBy` method of `Modules` is used, which searches for a specific module in a map by name, and if the name is not found, it loads the new one. Once this operation is completed, the next step involves extracting the `Method` on which to work from the module, and this is done using the line of the vulnerability (line 17). Finally, the `Statement` is extracted starting from the module. This operation concludes with a search for all the variables (inside the `Statement`), which are then collected and saved in a list of `Expression` objects. The choice here is `Expression` and not `NameExpr` because it should be noted that variable declarations have only `VariableDeclarationExpr`. So, in the case of a line like "`int a = b+3`", if only `NameExpr` were extracted, we would only get "`b`" without "`a`". For this reason, the filtering operation selects both `NameExpr` and `VariableDeclarationExpr`. Actually, it would be possible to cast a `VariableDeclarationExpr` into a `NameExpr`, but doing so would result in the loss of information about its origin in the code, such as its `Range`. Once this is done, all these expressions are used to initialize a new class called **`Variable`** (which will be discussed in the upcoming sections).

On the other hand, speaking of the `SQLInjection` class, let's begin by saying that it contains a variable inside it that defines its type: either **`DEFAULT`** or **`PARTIAL`**.

This is solely to understand whether we are dealing with a vulnerability of type `Statement` or `PreparedStatement`, respectively. In the latter case, there are many other factors to consider, including the fact that the user might have already defined some "setters", and these need to be found and placed correctly. Instead of providing readers with all the variables present in this class, we prefer to describe only the main ones necessary for initialization (the others will be introduced as needed during the explanation). The following is the code of the constructor for this class:

```
public class SQLInjection extends Vulnerability{
  enum SQL_INJ_VULNERABILITY{
    DEFAULT,
    PARTIAL
  }
  private Parameters params;
  private NameExpr queryVariable;
  private ExpressionStmt exprStmt;
  private String leftPart = "";

    ...

  private List<Replacement> replacements;
  private SQL_INJ_VULNERABILITY sqlVulnerability;
  private String[] indexNames = {"index", "i", "indexParameters", "
  indexValue"};
  private String index;

  private boolean acceptingReplacement = true;

  public SQLInjection(int line, String location, Modules modules)
  throws IOException {
    super(line, location, modules);
    Expression expr;
    if(this.stmt.isReturnStmt()) {
      expr = this.stmt.asReturnStmt().getExpression().get();
    }else {
      this.exprStmt = this.stmt.asExpressionStmt();
      expr = this.exprStmt.getExpression();
    }
    if(this.exprStmt!= null &&(expr.isVariableDeclarationExpr() ||
  expr.isAssignExpr())) {
      this.leftPart = JavaParserUtil.extractLeftPartFromExprAsString(
  this.exprStmt);
      this.variables.remove(0);
    }
    this.replacements = new LinkedList<>();

    this.queryVariable = this.variables.remove(0).asNameExpr();
    //If clearParameters is used then need to reconstruct for all the
     setters
    startIndexSearch();
    this.params = new Parameters(this);
    this.params.startRecursiveFind();

    if(this.sqlVulnerability == SQL_INJ_VULNERABILITY.PARTIAL)
      startSearchForSetters();
  }
}
```

List of Codes 5.14: SQLInjection constructor

As you can see, we immediately perform a check on the type of `stmt`, inherited from `Vulnerability`, in order to proceed with the extraction of the Expression present in it. Once this is done, the left side is "extracted" using a function defined in a module called `JavaParserUtil`. Then, the first variable is removed from the list of variables (remember that the first one on the left has already been saved). A list of **`Replacements`** is initialized; these are the types responsible for the code changes to be made. They have an `original` (what's initially there) and a `replacement` (what should be substituted); we will go into detail later. At this point, we take the name given by the user to the `Statement` variable, for example, "statement", and save it into `queryVaraible`. Then, we proceed with the search for the name to be given to a variable that will be used for setting (e.g., index). The code seems interesting; here it is:

```
public void startIndexSearch() {
    //Find an available name for setter index
        if(this.method.hasSQLIndex()) index = this.method.getSQLIndex
    ();
        else {
          for(String n: indexNames) {
            if(!this.method.isThisNameVariable(n)) {
              index = n;
              this.method.setSQLIndex(index);
              break;
            }
          }
        }
        //If none found iterate until one is ok:
        //Ex: index_1, index_2, index_3, index_4...
        if(index==null) {
          int i = 1;
          do {
            index = indexNames[0]+"_"+i++;
          }while(this.method.isThisNameVariable(index));
        }
  }
```

List of Codes 5.15: Seaching for name to give to index

Once this phase is completed, the **`Params`** type is initialized, which represents the contents of the vulnerable query. For example, if the variable "`sql`" is passed to the "`execute`" method, `params` will only contain "`sql`", otherwise, it will be composed of various elements. This type is the one that triggers the recursive search in the code. This is evident from the fact that after its initialization, the **`startRecursiveFind`** method is called. Once this phase is completed, a search is performed for any new **`Setters`**, but only if the type of `SQLInjection` is "`PARTIAL`". This instruction will be explained shortly.

5.3.4 Core logic

Now it is time to talk about the core of the tool, the most relevant part: the **logic** behind. First, we need to revisit the algorithm mentioned at the beginning, at point 3. As you can see, it requires performing a recursive search within the code. To do this, it's necessary to develop a robust structure that is always capable of pinpointing the specific point we are referring to. However, when necessary, it should be able to view the general context, such as the vulnerability line or the various modules. To achieve this, all classes, in one way or another, must be able to access the vulnerability from which they originate, namely the `SQLInjection`. In particular, recursion occurs by exploiting three components:

1. **`Replacement`**, which is a line that needs to be replaced with another, and it has fields such as "`ExpressionStmt original`" and "`ExpressionStmt replacement`".

2. **`Part`**, each `Replacement` will have various types of parts inside it, for example: `stmt.execute("SELECT * FROM students WHERE id='"+studentId+"'")`. In this case, the parts will be "`SELECT * FROM students WHERE id='`", `studentId`, and finally, "`'`".

3. **`Variable`**, all parts that are not fixed `Strings` define variables. For example, in the previous case, the part `studentId` is a variable.

These three classes are intertwined with each other because whenever a Variable is found, other lines in the code where it is used are searched for, and for each line that requires replacement, a new Replacement is initialized. Parts are extracted from the Replacement, and the cycle continues.

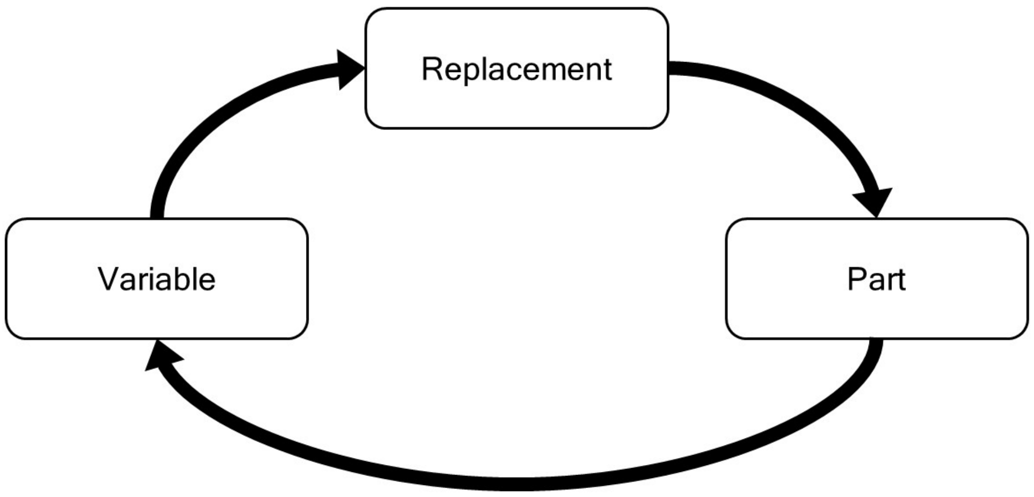


Figure 5.1: Schema of recursive call

5.3.5 Parameters class

The first class to be analysed will be the **`Parameters`** class. This is the one that triggers the entire recursive process, here there is the code:

```java
package it.reply.chatGPT;

import java.util.List;
import java.util.stream.Collectors;

public class Parameters {
  private List<Part> parts;
  private Replacement replacement;
  private SQLInjection sql;

  public Parameters(SQLInjection sqlInjection) {
    this.sql = sqlInjection;
    this.replacement = new Replacement(this.sql.getExpressionStmt(),
   this.sql);
  }

  public List<Part> getParts(){
    return this.parts;
  }
  public List<Part> getVariableStringParts(){
    return this.parts.stream().filter(Part::isString).collect(
   Collectors.toList());
  }

  public void startRecursiveFind() {
    this.replacement.startFind(this.sql.getSetter());
  }

  public void reconstructSetter(Setter setter) {
    this.replacement.startFind(setter);
  }

}
```

List of Codes 5.16: Parameters class

It contains a `List` of parts, the first `Replacement` and the reference to the `SQLInjection` (this is fundamental to be able to retrieve information about the whole project). As it can be seen from the code, in this case we are passing to the constructor of the `Replacement`, the original vulnerable `ExpressionStmt` and also a reference to the `SQLInjection`.

After that, the function **`startRecursiveFind()`** is invoked, it will be analysed later the reason why it is also passed a `Setter`.

5.3.6 Replacement first constructor

It must be highlighted that `Replacement` has more than one constructor. Now we will report the one invoked above, that is also the one used to initialized the first `Replacement`, the one referred to the vulnerable line. The code is now reported:

```
public Replacement(ExpressionStmt expressionStmt, SQLInjection sql) {
    this.original = expressionStmt;
  this.referedSQLVuln = sql;
  NameExpr rightFirst;
  if(this.original != null) {
    this.leftPart = JavaParserUtil.extractLeftPartFromExprAsString(
   this.original);
    this.rightPart = JavaParserUtil.extractRightPart(this.original);
    rightFirst = this.rightPart.findFirst(NameExpr.class).get();
  }else {
    rightFirst = this.referedSQLVuln.getStatement().asReturnStmt().
   getExpression().get().asMethodCallExpr().getScope().get().
   asNameExpr();
  }

  String varType;
  if(rightFirst.calculateResolvedType().isReference()) {
    varType = rightFirst.calculateResolvedType().asReferenceType().
   getTypeDeclaration().get().getName();
  }else if(rightFirst.calculateResolvedType().isConstraint()){
    varType = rightFirst.calculateResolvedType().asConstraintType().
   getBound().asReferenceType().getTypeDeclaration().get().getName();
  }else varType = "S";
  if (varType.equals("Connection")) {
    this.referedSQLVuln.setSqlInjVulnerability(SQL_INJ_VULNERABILITY.
   PARTIAL);
    handlePartialSQL();
  }else {
    this.referedSQLVuln.setSqlInjVulnerability(SQL_INJ_VULNERABILITY.
   DEFAULT);
    handleStandardSQL(rightFirst);
  }
}
```

List of Codes 5.17: Constructor of First Replacement

Basically we are extracting the right part, so the one that involves the `stmt`. It will be analysed the type and depending on it the `SQLInjection` type will be set (“`STANDARD`” or “`PARTIAL`”). At this point one of two functions is called. The first one is for the “`PARTIAL`” types:

```
private void handlePartialSQL() {
     Method m = this.referedSQLVuln.getMethod();
     NameExpr statementName;
     if(this.original.getExpression().isVariableDeclarationExpr()) {
          VariableDeclarationExpr vde = this.original.getExpression().
     asVariableDeclarationExpr();
          statementName = vde.getVariable(0).getNameAsExpression();
     }else {
          statementName = this.original.findFirst(NameExpr.class).get()
     ;
     }
     this.referedSQLVuln.setQueryVariable(statementName);
     this.parts = JavaParserUtil.splitString(this.rightPart.
     asMethodCallExpr().getArgument(0),
               this.referedSQLVuln.getMethod());
     this.replacement = this.original.clone();
     List<Statement> expressionStmtList = m.getDefinedSetters(this.
     original, statementName);
     this.referedSQLVuln.setDefinedSetters(expressionStmtList);
     this.referedSQLVuln.initForPartial();

     this.type = ExpressionType.NONE;
}
```

List of Codes 5.18: Partial Type

In this code we are getting the name of the statement variable (i.e.,`stmt`). We are splitting each parts using the code seen for **StreamEx**. After that, there is the extraction of all the user defined setters, because after a `PreparedStatement` the user must instruct the code on how to replace each placeholder "?". Then, a useful initialization is done inside the referred `SQLInjection`. What concerns the `type` will be explained in the recursive part of the Replacement. Before going on, the two methods will be analysed.

```java
public void setDefinedSetters(List<Statement> userSetter) {
    this.userSetterList = new ArrayList<>();
    LinkedList<Statement> subList = new LinkedList<>();
    for(Statement stmt: userSetter) {
        if(stmt.isExpressionStmt()) {
            if(stmt.asExpressionStmt().getExpression().
   isMethodCallExpr()){
                MethodCallExpr mce = stmt.asExpressionStmt().
   getExpression().asMethodCallExpr();
                if(mce.getNameAsString().equals("clearParameters")) {
                    this.userSetterList.add(subList);
                    subList = new LinkedList<>();
                    continue;
                }
            }
        }
        subList.add(stmt);
    }
    if (!subList.isEmpty()) this.userSetterList.add(subList);
    if (!this.userSetterList.isEmpty()) swithNewUserSetter();
}
```

List of Codes 5.19: Extraction of all setters

In case of a `PreparedStatement` we could find more then one execute, this because of the usage of the function `clearParameters()`. In this case the tool will look for this specific statement and, in case it is present, multiple lists will be instanciated. On the other hand, the other function is this one.

```java
public void initForPartial() {
    this.setterList = new LinkedList<>();
    this.setterAlreadyDone = new HashMap<>();
    this.setterAlreadyReplaced = new HashMap<>();
}
```

List of Codes 5.20: Additional initialization.

It is just the creation of new Maps and Lists that will be used later.

Now, let's move on to the other function in case the `SQLInjection` is "`STANDARD`":

```java
private void handleStandardSQL(NameExpr name) {
    // The original statement is something like:
    // Result r = statement.execute(query)
    this.type = ExpressionType.NONE;
    Method m = this.referedSQLVuln.getMethod();
    NameExpr nameDeclaration;

    // Importing the dependencies if needed
    this.referedSQLVuln.getModule().checkImport("java.sql.PreparedStatement");

    // Statement statement = connection.createStatement()
    VariableDeclarationExpr statement = m.findDeclarationStmt(this.original, name);
    if (statement == null) throw new IllegalStateException("Illegal state");

    // Removing the vulnerable statement from the list of the variable declaration
    // In case the list doesn't exist it will be created
    m.updateDeclarationMap(statement, original);
    Optional<NameExpr> variableName = m.getNewDeclarationInScope(statement, this.original);

    // Getting: statement.execute(...)
    MethodCallExpr mce = this.rightPart.findFirst(MethodCallExpr.class).get();
    // Splitting parts from execute(List<Part>)
    this.parts = JavaParserUtil.splitString(mce.getArgument(0), m);
    List<Expression> arguments = mce.getArguments().subList(1, mce.getArguments().size());
    // Change: statement.execute(List<Part>)
    // To: connection.prepareStatement(List<Part>)

    //First getting: connection.createStatement()
    MethodCallExpr declarationMethodCall;
    if(statement.getVariable(0).getInitializer().get().isNullLiteralExpr()) {
        List<ExpressionStmt> assignStmts = m.findAssignments(statement.getVariable(0).getNameAsString(), this.original.getBegin().get().line);
        ExpressionStmt assignStmt = assignStmts.get(assignStmts.size()-1);
        declarationMethodCall = assignStmt.getExpression().asAssignExpr().getValue().clone().asMethodCallExpr();
        m.addStmtToDelete(assignStmt);
    }else {
        declarationMethodCall = statement.getVariable(0).getInitializer().get().clone().asMethodCallExpr();
    }

    //Changing to: connection.prepareStatement()
    declarationMethodCall.setName("prepareStatement");

    //Adding argument: connection.prepareStatement("query")
    arguments.addAll(declarationMethodCall.getArguments());
    arguments = arguments.stream().distinct().toList();
    for(Expression arg : declarationMethodCall.getArguments()) {
        declarationMethodCall.getArguments().remove(arg);
    }
    System.out.println(declarationMethodCall);
    if(declarationMethodCall.getArguments().isEmpty()) {
        declarationMethodCall.addArgument(Part.reconstructor(this.parts));
    }else {
        declarationMethodCall.setArgument(0,Part.reconstructor(this.parts));
    }
    for(Expression arg: arguments) {
        declarationMethodCall.addArgument(arg);
    }
    if (variableName.isPresent()) {
        nameDeclaration = variableName.get();
        if(!m.variableDeclarationHasDependencies(statement)) {
            nameDeclaration.setName(statement.getVariable(0).getNameAsString());
        }
        //Assigning: statement = connection.prepareStatement("query")
        AssignExpr assignReplace = new AssignExpr(nameDeclaration, declarationMethodCall, Operator.ASSIGN);
        this.replacement = new ExpressionStmt(assignReplace);
    } else {
        if(!m.variableDeclarationHasDependencies(statement)) {
            nameDeclaration = statement.getVariable(0).getNameAsExpression();
        }
        else {
            int i = 0;
            while (m.isThisNameVariable(statement.getVariable(0).getNameAsString() + "_" + ++i));
            nameDeclaration = statement.getVariable(0).getNameAsExpression();
            nameDeclaration.setName(statement.getVariable(0).getNameAsString() + "_" + i);
        }
        m.addSQLName(this.original, nameDeclaration);
        //Adding name to the map of SQL initialization
        m.addVariableForSQL(statement, this.original, nameDeclaration);
        //Initializing: PreparedStatement statement = connection.prepareStatement("query")
        VariableDeclarationExpr newDeclaration = new VariableDeclarationExpr(
                new VariableDeclarator(
                        StaticJavaParser.parseType("PreparedStatement"),
                        nameDeclaration.getName(),
                        declarationMethodCall));
        this.finalName = nameDeclaration;
        this.replacement = new ExpressionStmt(newDeclaration);
    }

    this.referedSQLVuln.setQueryVariable(nameDeclaration);
    this.referedSQLVuln.setInitPhase(m.getInitUntilStmt(statement, this.original));
    //Saving inside the vulnerability
    this.referedSQLVuln.addReplacement(this);
}
```

List of Codes 5.21: Standard Type.

As you can observe, this method is considerably more complex than the previous one. This is primarily because no assumptions can be made. The variable declaration must be altered, and it is possible that other variables will depend on it, necessitating the preservation of dependencies.

The procedure begins by checking the imports to determine whether `PreparedStatement` is imported. Subsequently, the search for the `Statement` declaration is carried out, followed by the storage of all the dependencies within the `Method` class. At this point, an examination of the name is undertaken to ascertain if an alternative name is available or if a new one is required **(please refer to the variable definitions in the `Method` class for further clarification)**.

In any case, it is known that a `MethodCallExpr` (`connection.createStatement(...)`) will be present, and it will be transformed into a `PreparedStatement`. Initially, the method name is altered to `prepareStatement`, and subsequently, the parameters passed to the method are adjusted to preserve the previous initializations and incorporate the SQL query. Upon completing this process, an evaluation of the variable name ensues. If the name was previously in use, it is retained; otherwise, the new name is saved along with a reference to the corresponding `ExpressionStmt`. In both scenarios, this `ExpressionStmt` is stored in the `replacement` field (the `Expression` could be an `AssignExpr` in the event of an existing name, or a `VariableDeclarationExpr` in the case of a new name).

The transformation is readily apparent, as we transition from a construct like "`statement.execute(sql)`" to the form "`connection.prepareStatement(sql)`". The execution part will be reintegrated subsequently.

5.3.7 Replacement second constructor

Another important method of this class that need to be discussed is the second constructor. Indeed, the first one is use only once to analyse the vulnerable line, all the other times it is used the following one.

```java
public Replacement(ExpressionStmt expr, SQLInjection SQLVuln,
    Variable variable, MethodCallExpr currentMce) {
     this.original = expr;
     this.referedSQLVuln = SQLVuln;
     this.variable = variable;
     if(currentMce!= null) {
         this.limiter = JavaParserUtil.getExpressionStmtFromExpression
    (currentMce);
         this.currentMce = currentMce;
     }
     if (this.variable.isOutOfScope())
         this.isInVulnScope = false;
     else {
         isInVulnScope = JavaParserUtil.haveScopeWithoutLoop(expr,
    this.limiter == null ? SQLVuln.getExpressionStmt() : this.limiter)
    ;
     }
     if (original.getExpression().isVariableDeclarationExpr()) {
         this.leftPart = JavaParserUtil.
    extractLeftPartFromExprAsString(original);
         this.rightPart = JavaParserUtil.extractRightPart(original);
         this.type = ExpressionType.VARIABLE_DECLARATOR;
     } else if (original.getExpression().isAssignExpr()) {
         this.leftPart = JavaParserUtil.
    extractLeftPartFromExprAsString(original);
         this.rightPart = JavaParserUtil.extractRightPart(original);
         this.type = ExpressionType.ASSIGN_EXPRESSION;
     } else {

         this.type = ExpressionType.METHOD_CALL;
     }

     if (!leftPart.isEmpty())
         toRewrite = leftPart + " = ";

     if (this.type == ExpressionType.METHOD_CALL) {
         this.initReplacementForMethodCall();
     } else if (this.type == ExpressionType.ASSIGN_EXPRESSION) {
         this.initReplacementForAssign();
     } else if (this.type == ExpressionType.VARIABLE_DECLARATOR) {
         this.initReplacementForDeclarator();
     }

    if (this.variable.partHasFather()) {
        Part father = this.variable.getFatherPart();
        this.parts.forEach(p -> {
            p.setFather(father);
            p.setBranch(this.variable.getPartBranch());
        });
    }
}
```

List of Codes 5.22: Constructor of other Replacements.

The first part is there to handle strange cases, the one with `ConditionalExpr`. Then the statement is studyed and differently from the first case, there are three main possibilities:

- `MethodCallExpr`, the original `ExpressionStmt` is a method call (e.g., `sql-.append("SELECT * FROM student WHERE id='" + studentId + "'")`).
- `AssignExpr`, the original `ExpressionStmt` is just an assignment (e.g., `sql= "SELECT * FROM student WHERE id='" + studentId + "'"`).
- `VariableDeclarationExpr`, the original `ExpressionStmt` is a new variable (e.g., `String sql="SELECT * FROM student WHERE id='" + studentId + "'"`).

Once recognized the type, it is saved in a variable "`ExpressionType type`" and then the initialization is made using the three method that we can see in the code:

- **initReplacementForMethodCall()**;
- **initReplacementForAssign()**;
- **initReplacementForDeclarator()**;

These functions do almost the same thing, they must extract all the `Parts` from the original statement and create the replacement. The replacement is not "final", it means that we could change it, this will be clear after the discussion of the class `Part`.

The last few lines are there to assign to the `ConditionalExpr` the correct father and branch.

5.3.8 Replacement recursive

Now let's move into the main part of the `Replacement`, the method **startFind()**. It follows the code.

```java
public void startFind(Setter s) {
    this.parts.forEach(p -> p.startRecursiveFind(this.referedSQLVuln,
    s, this.currentMce));
    //If we don't have parts it is useless to go up
    if(this.parts.isEmpty()) return;
    if (this.type == ExpressionType.METHOD_CALL) {
        this.instanciateReplacementForMethodCall();
    } else if (this.type == ExpressionType.ASSIGN_EXPRESSION) {
        this.instanciateReplacementForAssign();
    } else if (this.type == ExpressionType.VARIABLE_DECLARATOR) {
        this.instanciateReplacementForDeclarator();
    } else if(this.type == ExpressionType.NONE) {
        this.replacement.findFirst(MethodCallExpr.class).get().
   setArgument(0, Part.reconstructor(parts));
    }
    if (this.referedSQLVuln.isAcceptingReplacement() && !this.
    original.toString().equals(this.replacement.toString()))
        this.referedSQLVuln.addReplacement(this);
}
```

The first thing that is done is to iterate over the parts, after that we need to fix the `replacement`, because during the research inside the code we could have changed the type of some `Parts`. Taking the case in which the type is "None", it is the case of the first Replacement, the one instanciated in the first constructor, we need to call `Part.reconstructor()`. Finally, we save the replacement if the `original` is different from `replacement`.

5.3.9 Part

This is another significant class. In order to understand the logic behind `Part .reconstructor()`, it must first be studied its constructor.

```java
public class Part {

    enum Type{
        FIXED,
        FINAL,
        STRING,
        CONDITIONAL,
        OUT_OF_SCOPE
    }

    ...

    private Expression fullExpr;
    private Expression current;
    private String part;
    private Type type;
    private Variable var = null;
    private Expression condition;
    private List<Part> ifClauses;
    private List<Part> elseClauses;
    private Part father;
    private Branch branch;

  public Part(Expression fullExpr, Expression expr, Method method) {
    this.fullExpr = fullExpr;
    this.part = expr.toString();
    Optional<NameExpr> ne = expr.findFirst(NameExpr.class);
    Expression parsedExpr = JavaParserUtil.exctractConditionalExpr(expr);
    this.current = parsedExpr;
    if(ne.isPresent()) {

      this.var = new Variable(ne.get(), this, method);

      if(this.var.isOutOfScope() && JavaParserUtil.isStringOrEquivalent(parsedExpr.calculateResolvedType())) {
        this.type = Type.OUT_OF_SCOPE;
      }
      else if(parsedExpr instanceof ConditionalExpr) {
        ConditionalExpr condExpr = parsedExpr.asConditionalExpr();
        this.type = Type.CONDITIONAL;
        this.condition = condExpr.getCondition();
        this.ifClauses = JavaParserUtil.splitString(condExpr.getThenExpr(), method);
        this.elseClauses = JavaParserUtil.splitString(condExpr.getElseExpr(), method);

      }
      else if(JavaParserUtil.isStringOrEquivalent(parsedExpr.calculateResolvedType())) {
        if(this.var.isOutOfScope()) {
          this.type = Type.OUT_OF_SCOPE;
        }else {
          this.type = Type.STRING;
        }
      }
      else this.type = Type.FINAL;
    }else this.type = Type.FIXED;
  }
}
```

List of Codes 5.23: Part constructor

This constructor accepts the complete `Expression`, the specific one we are referring to, and the `Method` it is located in. By process of elimination, a type is assigned to the `Part`, which can be:

1. "**FIXED**", defined in the code, immutable.
2. "**FINAL**", user-derived, to be replaced in the statement.
3. "**CONDITIONAL**", i.e., with a condition.
4. "**STRING**", of string type.
5. "**OUT_OF_SCOPE**", outside of the method.

The first type is assigned to all `StringLiteralExpr`. Initially, "FINAL" is only assigned to non-string variables. Therefore, all the latter will be labeled as "STRING", and a search is performed in the various calling methods to determine whether they actually contain SQL code or are user-derived. Finally, work is also done on the `CONDITIONAL` type, from which the sides of the "if" statement and the "else" statement are extracted (even in the case of recursion). Now it is possible to understand the function **startRecursiveFind()** invoked by the `Replacement`. In fact, in the init phase of `Replacement`, this function creates the `Parts` that, in turn, create the `Variables`, but the actual search is triggered when this method is called.

```java
public void startRecursiveFind(SQLInjection sqlVuln, Setter setter,
    MethodCallExpr currentMce) {
    int indexWhereToAdd = setter.getIndex();
    boolean wasString = false;
    if(this.type == Type.STRING) {
        this.var.startFind(sqlVuln, setter, currentMce);
        wasString = true;
    }else if(this.type == Type.OUT_OF_SCOPE) {
        TypeOutOfScope out= this.var.startFindOutOfScope();
        if(out == TypeOutOfScope.USER) this.type = Type.FINAL;
        else throw new IllegalStateException("Unable to fix because
    invoker of this method have insecure SQL, the tool is not able to
    change methods' prototypes");
    }

    if(this.type == Type.CONDITIONAL) {
        handleStartRecursiveConditional(sqlVuln, setter, currentMce);
    }else if(this.type == Type.FINAL && wasString) {
        addSetToSetter(setter, sqlVuln, indexWhereToAdd);
    }else if(this.type == Type.FINAL) {
        addSetToSetter(setter, sqlVuln);
    }else if(this.type == Type.OUT_OF_SCOPE) {
        addSetToSetter(setter, sqlVuln);
    }else if(sqlVuln.getType() == SQL_INJ_VULNERABILITY.PARTIAL){
        handlePartialSQLFixed(sqlVuln, setter);
    }
}
```

List of Codes 5.24: startRecursiveFind() of Part

It is evident from the code that if the variable type is a String, a search is performed within the method. If it is of type 'OUT_OF_SCOPE', it is searched in the calling methods. As for the first two methods, they will be discussed when we study the `Variable` class. However, it is essential to clarify that both can change the type of the `Part`.

Now, let's explain what happens from `line 13` onwards. Starting with the first case, it is nothing more than a wrapper to recursively call `startRecursiveFind` but using the "if" or "else" as the part.

The other cases add the respective `Part` to the setters. In the case with "`indexWhereToAdd`", the position in the setter is also specified, while in the second case, an append operation is performed in the list of setters. To understand what happens in the `setter`, it is needed to delve into how the `Setter` class works.

The last case is the most interesting and is the one we will focus on. In the case of a partially vulnerable query, there could be several scenarios. First of all, we need to divide the code into two parts: the initialization of the `PreparedStatement`, where the query is passed and the various "?" placeholders are positioned, and the setting part, where the variables are assigned values, i.e., how to replace each previously inserted placeholders.

These two code sections can also have different loops. For example, for a "?", it might assigned through an "if" statement. In general, we distinguish three cases depending on these code sections:

1. They have the same branches. This is the simplest case, where each "?" corresponds to a single setting statement.

2. The "?" has more branches than the setting phase. This is the case where we have conditions (if-then-else) above the "?" where one condition executes one type of query, and another condition executes a different type of query. However, in the end, there is always only one "?" to set.

3. The setters have more branches than the "?". In this case, the setting phase can have an `if-then-else`, while above, there is only one "?".

These cases can be further complicated by the presence of multiple placeholders in the same statement. This function's task, where possible, is to reconstruct the entire setting phase and insert each setter in the right place in the setters. We won't analyse its details, but case 1 is handled simply at the beginning of the function; while there are two other methods, **handlePartMoreThanSetter** and **handlePartLessThanSetter**, which handle cases 2 and 3. Note that in these methods, it is very easy to throw an exception because it is always preferred to avoid correction rather than add non-working code.

Now that we've understood how the `Part` class works, let's take a look at the **reconstructor**. First, it searches for delimiters that can be either “"” or “'”. Once this is done, the parts of type “FINAL” are replaced with the placeholder “?” (because at this point, the search within the code will already be completed). Below is the main code.

```java
for(Part p: parts) {
    if(p.type == Type.FIXED) {
        if(!isOpenFixed) {
            concat += delimiterFixed;
            isOpenFixed = true;
        }

        concat += p.getWithoutQuote();
    }else if(p.type == Type.FINAL) {
        if(!isOpenFixed) {
            concat += delimiterFixed;
            isOpenFixed = true;
        }
        concat += "?";
    }else if(p.type == Type.CONDITIONAL) {
        if(isOpenFixed) {
            concat += delimiterFixed+" +";
            isOpenFixed = false;
        }
        concat += " ("+ p.condition + " " +placeHolder+" ";
        concat += reconstructor(p.ifClauses) + " : ";
        concat += reconstructor(p.elseClauses) + ") +";
    }else {
        if(isOpenFixed) {
            concat += delimiterFixed;
            isOpenFixed = false;
        }
        concat += "+ " + p.part + " +";
    }
}
if(concat.startsWith("+")) concat = concat.substring(1);
if(concat.endsWith("+")) concat = concat.substring(0, concat.length()-1);
else concat += delimiterFixed;
```

List of Codes 5.25: Part recontructor.

Actually, several checks are subsequently performed to ensure that the replacement is done correctly, with special attention to everything inside the delimiters to avoid losing important pieces. For example in cases where we have something like: `sql="SELECT * FROM students WHERE dateEnroll = '" + date + "_00:00:00'"`, the last part about the time is fundamental and so it must be inserted in the setter, the whole statement needs to become this one: `sql="SELECT * FROM students WHERE dateEnroll = ?"`.

5.3.10 Variable

Variable can also have multiple constructors depending on the caller. However, unlike `Replacement`, its behavior remains the same. When other constructors are called this class will only have additional information, but behind will call the first constructor. This one will infer the type of the `Variable` and save the `Expression` to which it refers.

Much more interesting is the operation of the two functions we've seen in `Part`: **startFind()** and **startFindOutOfScope()**. Despite being profoundly different, both have the task of searching for every line that uses this `variable`. The first does this within the method, while the second does it externally.

The "startFind" function is responsible for finding the lines of code that use the variable previously. Once this is done, it initializes new Replacements if needed (there is a dedicated method in `Replacement` that returns a boolean: `Replacement.needReplacement(Statement)`). However, to understand the functioning of this method, it's necessary to anticipate that Setters need to know the variable's branch (if, while, for) and thus its scope to work. For each of these, a new Setter is required. For this reason, in this method, it will be found a map used to associate a `Statement` with a particular Branch.

It's important to note that `JavaParser` doesn't always work perfectly with "`if_then_else`" constructs. To illustrate, consider the following code example:

```java
if (startDate != null && startDate.length() > 0 && endDate != null &&
     endDate.length() > 0) {
    dateString = "DATE BETWEEN '" + startDate + " 00:00:00' AND '" +
  endDate + " 23:59:59'";
} else if (startDate != null && startDate.length() > 0) {
    dateString = "DATE > '" + startDate + " 00:00:00'";
} else if (endDate != null && endDate.length() > 0) {
    dateString = "DATE < '" + endDate + " 23:59:59'";
}
```

List of Codes 5.26: If-then-else from NotSecureBank [68]

In this case, `JavaParser` will interpret this entire `Node` as an `IfStmt`. This means that it also includes the "else if" and "else" inside it. To access the "child" that is actually the "else if", it is needed to invoke **getElseStatement()**, which returns the following result.

```java
if (startDate != null && startDate.length() > 0) {
    dateString = "DATE > '" + startDate + " 00:00:00'";
} else if (endDate != null && endDate.length() > 0) {
    dateString = "DATE < '" + endDate + " 23:59:59'";
}
```

List of Codes 5.27: Child of the previous if.

To resolve this issue, it was decided to distinguish two different maps: one for all `Statements` and another one for `if_then_else` constructs. In practice, the first map contains only the parent `if_then_else` that encompasses the others. Branches are associated with Setters in these two maps. If a branch is in the map, the Setter to use is that one. Otherwise, a new Setter must be created and added to this map. Below is the code for this function.

```java
public void startFind(SQLInjection SQLVuln, Setter setter,
    MethodCallExpr currentMce) {
    Setter actualSetter = setter;
    Map<Node, Setter> setterMap;
    Map<Node, Setter> ifSetterMap = new HashMap<>();
    Map<ExpressionStmt, Node> exprMap;

    if(this.method == null) {
        this.method = SQLVuln.getMethod();
        this.setIsOutOfScope();
    }

    this.statements = this.method.findAssignments(this.variable.
    getNameAsString(), lastValidLine);

    if(this.statements == null) {
        this.referedPart.setType(Type.FINAL);
        return;
    }
    ExpressionStmt sink = SQLVuln.stmt.asExpressionStmt();

    if(currentMce != null && !SQLVuln.getMethod().
    getMethodDeclaration().equals(JavaParserUtil.getMethodDeclaration(
    currentMce))) {
        this.replacements = this.statements.stream()
                .map(stmt-> new Replacement(stmt, SQLVuln, this,
    currentMce))
                .toList();
        sink = JavaParserUtil.getExpressionStmtFromExpression(
    currentMce);
    }else {
        this.replacements = this.statements.stream()
                .filter(stmt-> Replacement.needReplacement(stmt))
                .map(stmt-> new Replacement(stmt, SQLVuln, this,
currentMce))
                .toList();
    }

    if(this.replacements.isEmpty() && this.statements.stream().
noneMatch(stmt-> JavaParserUtil.containsSQLCode(stmt))) {
        this.referedPart.setType(Type.FINAL);
        return;
    }

    final ExpressionStmt toUse = sink;
    //It is fundamental to keep found order, the LinkedHashMap is
needed
    exprMap = this.replacements.stream()
    .collect(Collectors.toMap(
            r-> r.getOriginal(),
            r-> JavaParserUtil.getBiggerNodeBeforeSink(r.getOriginal
(), toUse),
            (a,b)->a,
            LinkedHashMap::new
            ));

    //Create a map with a <Node, Setter>
    setterMap = exprMap.values()
            .stream()
            .distinct()
            .collect(
                    Collectors.toMap(
                            node -> node,
                            node -> Setter.createFromBranch(node,
setter),
                            (a,b) -> a,
                            LinkedHashMap::new
                            ));
    //For each setter add it to the parent
    setterMap.values().forEach(actualSetter::addSetter);

    this.replacements.forEach(r->{
        Map<Node, Setter> referedMap = setterMap;
        Node keySetter = exprMap.get(r.getOriginal());
        Setter s = setterMap.get(keySetter);
        List<Statement> branches = JavaParserUtil.extractBranch(r.
getOriginal(), keySetter);
        if(!branches.isEmpty() && !branches.get(0).isIfStmt())
branches.remove(0);
```

```java
        if(!branches.isEmpty()) branches = JavaParserUtil.mergeIfStatement(branches);

        for(Statement branch : branches) {
            Setter support;
            if((branch.isIfStmt() || branch.isBlockStmt()) && referedMap != ifSetterMap) referedMap = ifSetterMap;
            if((!branch.isIfStmt() && !branch.isBlockStmt()) && referedMap != setterMap) referedMap = setterMap;

            if(referedMap.containsKey(branch)) {
                support = referedMap.get(branch);
            }else {
                support = Setter.createFromBranchesNotFirst(branch,s);
                s.addSetter(support);
                referedMap.put(branch, support);
            }
            s = support;
        }
        r.startFind(s);
    });

    if(this.replacements.stream().allMatch(r-> r.getOriginal().equals(r.getReplacement()) && !JavaParserUtil.containsSQLCode(r.getOriginal()))) {
        this.referedPart.setType(Part.Type.FINAL);
    }
}
```

List of Codes 5.28: startFind() of Variable.

On the other hand this is the code for “startFindOutOfScope”:

```java
public TypeOutOfScope startFindOutOfScope() {
    Parameter p = this.method.getMethodDeclaration().getParameterByName(this.variable.toString()).get();
    int index = this.method.getMethodDeclaration().getParameters().indexOf(p);
    if(this.method.getMethodDeclaration().getAnnotations().stream().anyMatch(a-> a.getNameAsString().endsWith("Mapping"))) return TypeOutOfScope.USER;
    Map<Method, List<MethodCallExpr>> caller = this.method.getModule().getModules().getCallers(this.method.getMethodDeclaration());
    List<VariableExplorer> variables = new ArrayList<>();

    for(Entry<Method, List<MethodCallExpr>> e : caller.entrySet()) {
        Method m = e.getKey();
        for(MethodCallExpr mce : e.getValue()) {
            Expression expr = mce.getArgument(index);
```

```
                Part part = new Part(mce, expr, m);
                part.getVariable();
                VariableExplorer ve = new VariableExplorer(part.
    getVariable(), m);
                variables.add(ve);
            }
        }
        if(variables.isEmpty()) return TypeOutOfScope.USER;
        TypeOutOfScope result = null;
        for(VariableExplorer ve : variables) {
            TypeOutOfScope typeOut = ve.defineType();
            if(result == null) {
                result = typeOut;
            }
            if(typeOut==TypeOutOfScope.UNDEFINED) {
                continue;
            }
            if(typeOut != result) {
                System.out.println(typeOut);
                System.out.println(result);
                throw new IllegalStateException("Parameters assigned to
    the function are not coherent");
            }
        }
        if(result==TypeOutOfScope.OUT_OF_SCOPE || result ==
    TypeOutOfScope.UNDEFINED) return TypeOutOfScope.USER;
        return result;
}
```

List of Codes 5.29: startFindOutOfScope() of Variable.

The first step is to search for calling methods until the first caller is reached (which could be the `Spring Controller`). Once the methods are found, the `variable` is searched for in them, and it is ensured that everything is consistent and that the tool is indeed capable of fixing it. If it is discovered that sometimes the `variable` can contain SQL code and other times it cannot, an error is raised. Additionally, the tool cannot modify prototypes, so if such an operation is needed, the program will inform the user of its inability to perform the fix (again launching an exception) and will continue with other vulnerabilities.

5.3.11 Setter

In the previous sections, it has been mentioned several times that the **Setter** class is responsible for writing the setting phase. Although it is being discussed only now, this process occurs in parallel. Firstly, the structure of the Setter is that of a tree, where each node (by node, it is meant the one in the tree, not in `JavaParser`) can have from `0` to `n` child. Before showing an image describing its structure, below are the various types of setters:

- **ROOT**: The first Setter, the parent of all others.
- **FOR**, **WHILE**, **IF_THEN_ELSE**, **DO_WHILE**: Nodes that describe the type of node based on the branch.
- **IF_CLAUSE**, **ELSE_CLAUSE**, **ELSE_IF_CLAUSE**: Types that are found only when the parent is an `IF_THEN_ELSE`.
- **WRAPPER**: Used only to indicate a generic branch (e.g., a `BlockStmt`).
- **STMT**: Nodes where `ExpressionStmts` to be added are saved.

As can be seen from **`this code in the final part`**, a new Setter is defined for each branch. Subsequently, in the `Part`, there will be STMT-type Setters to save the various statements to be added later. Below are the fields of the Setter class.

```java
public class Setter {
  enum Type{
    FOR,
    IF_THEN_ELSE,
    WHILE,
    DO_WHILE,
    STMT,
    IF_CLAUSE,
    ELSE_CLAUSE,
    ELSE_IF_CLAUSE,
    ROOT,
    WRAPPER
  }
  private Part part;
  private Setter parent = null;
  private Statement block;
  private Type type;
  private NodeList<Expression> condition;
  private List<Variable> variables;
  private List<Replacement> replacements;
  private LinkedList<Setter> childs;
  private String varType;
  private List<ExpressionStmt> expressions = new LinkedList<>();
  private Map<Node, Setter> setterMap;
  private Node original;

    ...

}
```

List of Codes 5.30: Setter fields.

At this point it is convenient to show a picture showing a possible tree of the setters:

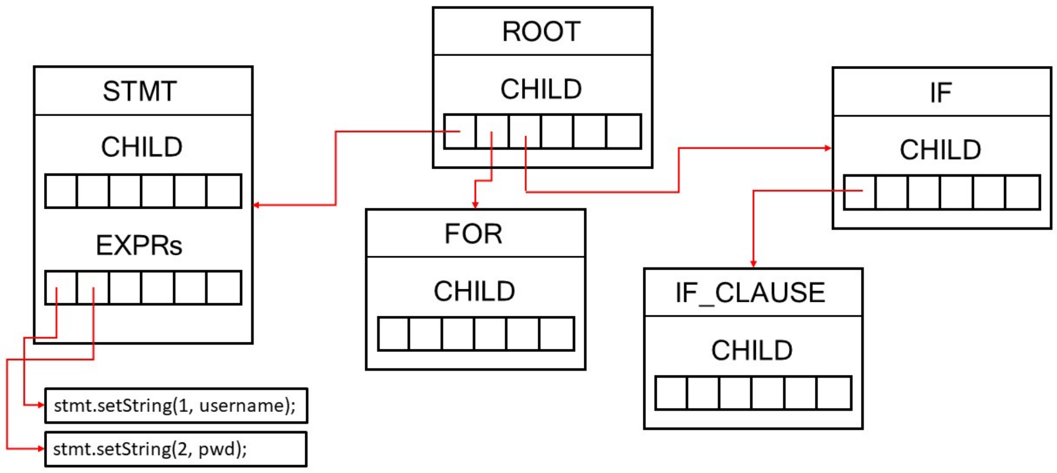


Figure 5.2: Example of a tree of setters.

One of the main functions of this class is **prepareSetter()**, which allows adding a Statement.

```java
public ExpressionStmt prepareSetter(Expression current, NameExpr stmt
    , String index) {
     ResolvedType rType = current.calculateResolvedType();
     String type;
     if(rType instanceof ResolvedArrayType) {
          type = rType.asArrayType().getComponentType().asReferenceType
     ().getTypeDeclaration().get().getName() +"[]";
     }
     else if (!rType.isPrimitive()) {
          type = rType.asReferenceType().getTypeDeclaration().get().
    getName();
     }else{
          type = rType.asPrimitive().describe();
          type = type.substring(0, 1).toUpperCase() + type.substring(1)
     ;
     }
     ExpressionStmt expr = new ExpressionStmt();
     MethodCallExpr mce = new MethodCallExpr();
     mce.setScope(stmt);
     mce.setName("set"+type);
    mce.addArgument(index+"++");
    mce.addArgument(current);

    expr.setExpression(mce);

    this.expressions.add(expr);
    return expr;
}
```

List of Codes 5.31: prepareSetter method.

First, the type is converted to a string, and then all the steps are taken to recreate the `statement`. It is noticeable how access to the SQLInjection variable is required to know how to call the `index`.

Another fundamental function is **reconstructSetter**, which starts from the root and explores the tree by descending from the leftmost children. Once the STMT-type leaves are reached, the statements are recreated and passed to the parent. This process is executed throughout the tree.

```
public String reconstructSetters() {
    String result= "";
    String partial = "";
    if(this.type == Type.STMT) {
        for(ExpressionStmt expr: this.expressions) result += expr +"\
  n";
        return result;
    }
    if(this.type == Type.IF_THEN_ELSE) {
        String tmp = "";
        List<Setter> useless = new ArrayList<>();
        List<String> statements = new ArrayList<>();
        for(Setter s: childs) {
            tmp = s.reconstructSetters();
            if(tmp.equals("")) useless.add(s);
            else statements.add(tmp);
        }
        return manageIfThenElseChild(statements, useless);
    }
    for(Setter s : childs) {
        partial += s.reconstructSetters();
    }
    if(partial.isEmpty()) return "";
    if(this.type == Type.FOR) {
        result += "for("+formattingCondition(this.condition) +"){\n";
        result += partial;
        result += "}\n";

    }else if(this.type == Type.DO_WHILE) {
        result += "do{\n";
        result += partial;
        result += "}while("+formattingCondition(this.condition)+")\n;
    ";
    }else if(this.type == Type.WHILE) {
        result += "while("+formattingCondition(this.condition) +"){\n
    ";
        result += partial;
        result += "}\n";
    }else return partial;
    return result;
}
```

List of Codes 5.32: reconstructSetter() method.

In the `SQLInjection` constructor (**here**) whenever the type is “PARTIAL”, we search again for the setters, this is done because whenever we have this type of `SQLInjection`, if there is a `clearParameters()`, we have no other choice then reanalyze the code and see what is needed to set.

5.3.12 Applying fixes

In order to apply all the fixes (replace in the replacement, adding new imports, adding and deleting variables in method), the class `VulnerabilityFixer` has the method `fix()` that will apply all the needed patches.

5.4 Input

As input, the tool accepts a report produced by SonarQube in CSV form. It doesn't mean that another SAST can not be used, but the header must be renamed accordingly to the values expected by the program. Here there are the name of the title accepted:

- **rule**. It is the name of the vulnerability according to FindSecBugs, it must contain the char sequence “SQL” and “injection” (it is not case sensitive). Furthermore, it can also contain the words “nonconstant” and “partial”.
- **component**. It indicates the location in the project. The various component must be splitted using “:”, that is the special character used by sonarQube, the last part must be the one indicating the name of the Java files, e.g., “`it.reply.test:test:src/main/java/util/DBUtil.java`”.
- **textRange**, it contains information regarding the position of the vulnerability inside the file. Again, it was choosen to use an approach like the one used by SonarQube, so with the starting line and ending line. e.g., “`{endLine:173,endOffset:85,startOffset:0,startLine:173}`”

Other columns are not required, but their presence will no alter the behaviour of the tool, indeed the tool will copy the entire entries in the report and will produce as result all the entries with an additional column, the one telling rather the operation was completed successfully or not.

5.5 Integration with Git

In this section, we will address the integration with Git that we mentioned in the dependencies section. To do this, it was used `JGit`. First, the user is asked for their username and password. Subsequently, the equivalent of the "`git status`" command is executed to check if there are any files to commit. If any, the user is asked if they want to commit the current state. Once this step is completed, a new branch is created in which the new `CompilationUnits` that have been fixed are saved. Finally, a commit is made, followed by a push.

After completing this operation, we differentiate between GitHub and GitLab. For the former, we use the GitHub API library to create a pull request, for the latter, we use HttpClient. GitLab assigns an ID to identify the project. So, if we only have the project's name, we first need to retrieve the ID. Once we have the ID, we can create the merge request. The above description is what happens in the following code.

```java
try {
    Status status = git.status().call();

    if(status.hasUncommittedChanges() || !status.getUntracked().
    isEmpty()) {
        if(!handleUncommitted(git)) {
            System.out.println("Exiting");
            return;
        }else {
            BufferedReader reader = new BufferedReader(new
    InputStreamReader(System.in));
            System.out.println("Commit comment: ");
            String comment = reader.readLine().strip();
            git.add().addFilepattern(".").call();
            git.commit().setMessage("VulnerabilityFixer: " + comment)
    .call();
        }
    }
    refName = repo.resolve(Constants.HEAD).getName();

    //Retrieving user information

    CredentialsProvider cp = new UsernamePasswordCredentialsProvider(
    username, pwd);
    //Try if push is working
    git.push().setCredentialsProvider(cp).call();

    opening = true;
    boolean branchSuccess = true;
    try {
        git.checkout().setCreateBranch(true).setName("
VulnerabilityFixer").call();
    }catch(RefAlreadyExistsException ref) {
        branchSuccess = false;
    }
    for(int i = 1; !branchSuccess; i++) {
        name = baseName + "_" + i;
        branchSuccess = true;
        try {
            git.checkout().setCreateBranch(true).setName(name).call()
;
        }catch(RefAlreadyExistsException ref) {
            branchSuccess = false;
        }
    }
    branchCreation = true;
    exportGit();
    git.add().addFilepattern(".").call();
    git.commit().setMessage("VulnerabilityFixer: fixed Java files
pushed").call();
    git.push().setCredentialsProvider(cp).call();
    if(typeRepo.equals("HUB")) {
        GitHub gitHubBuilder = new GitHubBuilder().withPassword(
username, pwd).build();
        // Repository information
        GHRepository repositoryGitHub = gitHubBuilder.getRepository(
owner + "/" + repoName);

        // Create a pull request
        GHPullRequest pr = repositoryGitHub.createPullRequest("
Automatic_fixing_vulnerabilities: pull request", name,
actualBranchName, "Automatic_Fixing_Vulnerabilities: merging new
branch with original one");

        System.out.println("Pull request created: " + pr.getHtmlUrl()
);

    }else {
        HttpClient client = HttpClients.createDefault();
        String gitLabApiUrl = "https://gitlab.com/api/v4";
        HttpGet request = new HttpGet(gitLabApiUrl+ "/users/" + owner
 + "/projects");
        request.addHeader("PRIVATE-TOKEN", pwd);
```

```java
        int projectId = -1;
        HttpResponse response = client.execute(request);
        String jsonResponse = EntityUtils.toString(response.getEntity());
        System.out.println("JSON: "+ jsonResponse);
        JSONArray projects = new JSONArray(jsonResponse);
        System.out.println(projects.length());
        for (int i = 0; i < projects.length(); i++) {
            JSONObject project = projects.getJSONObject(i);
            String projectName = project.getString("name");
            System.out.println(projectName);
            if (projectName.toLowerCase().equals(repoName)) {
                projectId = project.getInt("id");
                break; // Exit loop once the project is found
            }
        }
        if(projectId == -1) throw new NotFoundException("Project with name: '"+ repoName +"' not found");

        String mergeUrl = gitLabApiUrl + "/projects/" + projectId + "/merge_requests";
        HttpPost httpPost = new HttpPost(mergeUrl);
        httpPost.addHeader("PRIVATE-TOKEN", pwd);

        String jsonBody = "{\"source_branch\": \""+name+"\", \"target_branch\": \""+actualBranchName+"\", \"title\": \"Automatic_fixing_vulnerabilities: merge request\"}";

        StringEntity entity = new StringEntity(jsonBody);
        entity.setContentType("application/json");
        httpPost.setEntity(entity);

        response = client.execute(httpPost);

        if (response.getStatusLine().getStatusCode() == 201) {
            System.out.println("Pull request created successfully.");
        } else {
            System.err.println("Failed to create pull request. HTTP status code: " + response.getStatusLine().getStatusCode());
        }

        // Ensure to close the response
        HttpEntity responseEntity = response.getEntity();
        if (responseEntity != null) {
            responseEntity.getContent().close();
        }
    }
```

List of Codes 5.33: Git integration.

At the end of the process a file called “`Fixed_vulnerability.csv`” is stored. If something fails for some reason, everything will be saved in a folder called “`fixedFiles`”.

Chapter 6

Testing

In this chapter, it will be presented the various tested projects and highlighted the results achieved. Firstly, it will be explained how the tests were conducted, and then the testing projects will be analyzed, which will be examined both individually and in the context of other projects.

6.1 Method of testing

To conduct the tests, an initial attempt was made to use an automatic approach leveraging **JUnit**. Unfortunately, searching online, none of the found projects had these tests already implemented. For this reason, a **manual** approach was chosen, and the following steps were taken:

1. Obtaining the initial report from SonarQube.
2. Running the tool to fix vulnerabilities.
3. Manual code review, to check if the new code seems correct.
4. Building the fixed project, to be sure that syntactically everything was preserved.
5. Getting and comparing the new SonarQube report.
6. Performing manual tests on the launched project, to ensure that the vulnerable was fixed and that the code is semantically correct.

For each project these steps were executed. This approach was adopted to ensure that after having solved each vulnerability, the code is no longer vulnerable but also its behaviour is the same as before the fixing.

6.2 Projects

The tool was tested on a total of 9 profoundly different projects. This was done to ensure that the resolution of various vulnerabilities did not occur due to similarities between different vulnerabilities, but that everything worked on even dissimilar cases. To achieve this, projects were used from various sources, all from different authors. Some are real applications intentionally designed to be vulnerable, not only to SQL injection (although the program can only correct that); others are smaller applications that only have few vulnerabilities. In some cases, both the vulnerable and corrected versions were available, which clearly facilitated their testing. The various projects used are listed here with a brief description:

- **easybuggy** [69]: A Maven project intentionally made vulnerable and also containing SQL injections (Lines of code: 8.1k).
- **java-sec-code** [70]: Another Maven project with many vulnerabilities, including a corrected version of SQL (Lines of code: 7.4k).
- **javaSec-SQLInject-Demo** [71]: A small Maven application with some SQL injections, chosen for the presence of vulnerable PreparedStatements, thus testing the "PARTIAL" SQL tool (Lines of code: 2.8k).
- **JavaVulnerableLab** [72]: Also Maven-based, with several vulnerabilities including SQL (Lines of code: 4.1k).
- **NotSecureBank** [68]: A Gradle project used for the secure coding course, filled with vulnerabilities, including SQL injections (Lines of code: 9.1k).
- **SQLInjection Test** [73]: A small Maven project with a single SQL injection (Lines of code: 316).
- **tarpit-java** [74]: A few vulnerabilities in a deliberately vulnerable Maven project (Lines of code: 2.1k).
- **vulnado** [75]: Another Maven application with some vulnerabilities, including SQL (Lines of code: 1.9k).
- **WebGoat** [76]: A deliberately vulnerable Maven application maintained by OWASP, containing many vulnerabilities, especially those in the top 10 (Lines of code: 216.8k).

Totally, SonarQube identified 62 SQL Injection across these 9 projects. Not all of them were actually exploitable, so the following chart is provided for clarification:

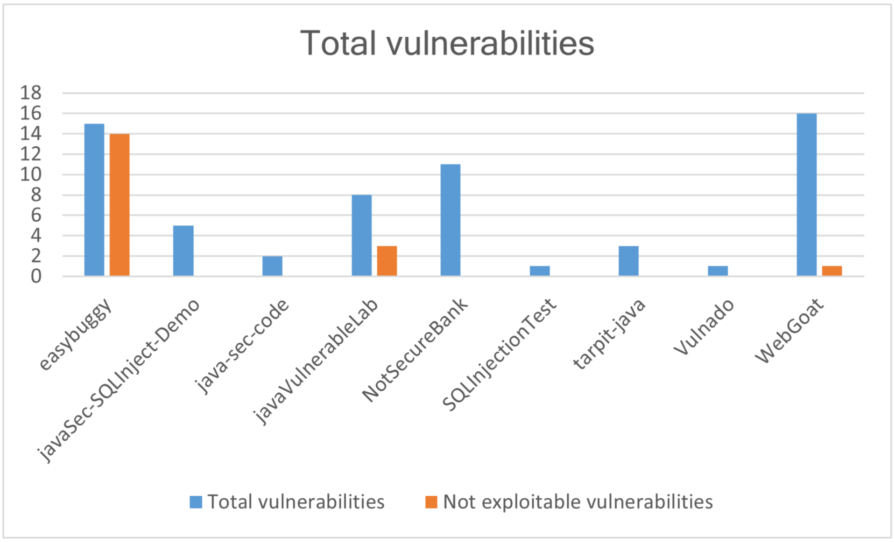


Figure 6.1: Total SQL Injection for each project

As it can be seen in the chart, there are some projects with more SQL Injections. It is fundamental to remark that inside each report produced by Sonar, there were much more vulnerabilities, but the tool is capable of filtering the SQL Injections. Considering that we have used 9 projects, having 62 vulnerabilities is a good result and all of them let us understand the capabilities of this tool. As it can be seen in here, among the 62 vulnerabilities, 44 are true positives, while 18 are actually not exploitable, in the next sections this results will be explained.

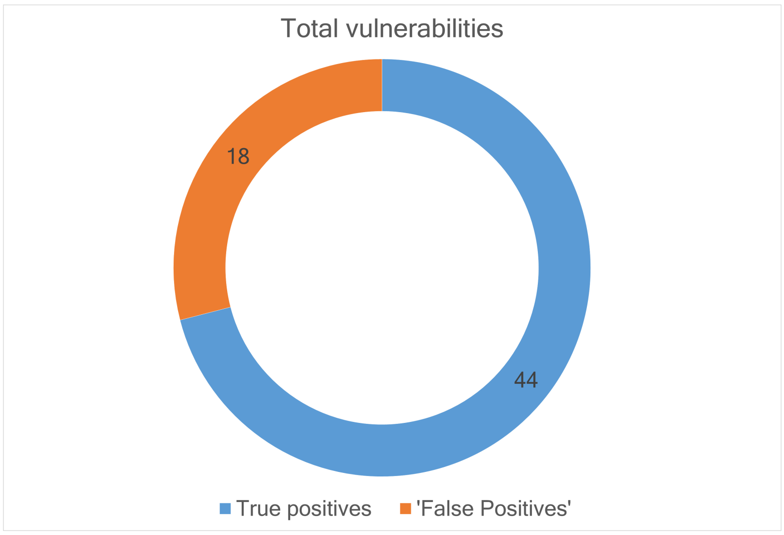


Figure 6.2: Distribution of the 62 vulnerabilities

6.3 True positive

True positives are all the cases where SQL Injection is actually exploitable, so it means that having them inside the code might lead to attacks. For this reason, they are for sure the most dangerous ones. Usually, in those projects it can be found a true positive whenever it is present a Statement that contains the keyword “SELECT”. Having other keywords doesn’t mean that the code is 100% secure, of course there are cases of insertion (i.e., “INSERT”) that were vulnerable to SQL Injection and the tool was capable of solving also them. The tool was also tested on `PreparedStatement`, this was done in order to be sure that it was actually working on those cases. Below there is a chart that describes the total amount of solved vulnerabilities among the 44 true positives:

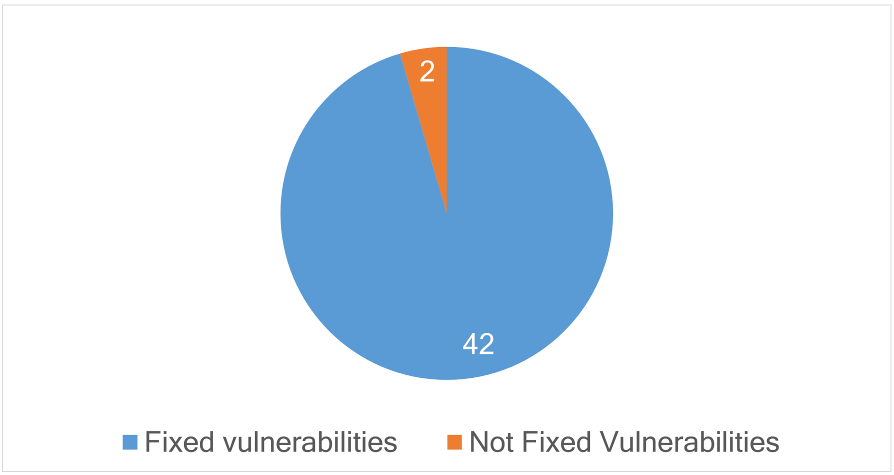


Figure 6.3: Pie chart of True positives

It is important that the tool is capable of fixing correctly, but it is more important that the fix doesn’t alter the behaviour of the program, for this reason whenever strange cases are encountered, the tool throws an exception explaining the reason, such as the two cases that were not fixed (these two cases will be explained below). This means that for 42 times, the corrections ensure that the code will be no longer vulnerable. Considering what was said before, this is a wonderful result, because it highlights that in most cases the tool is sure that applying the fixes, not only the code will no longer be vulnerable, but also that its semantic will remain unaltered.

Talking about the two cases that were not fixed, it must be analysed the background of this tool. Those projects were developed in order to be vulnerable and so to remain vulnerable. In other words: the best practice of developing are not respected. Now let’s analyse the details of each one:

- In one case of `WebGoat` we have the various “.java” files not well organized. As it was explained in the part regarding the accepted parameters, one of those (`src`) is useful to let the tool understand where are the code files inside the project directory, unfortunately if the directory is not ordered in this sense, there is a small possibility that the tool is not capable of inferring the type of some variables (because of the missing `jar`). This process is fundamental in Java, indeed to understand the call hierarchy of a method, all the other methods must be analysed. To be sure that the method under analysis is actually that one, it is not enough comparing the name, also the parameters must be of the same time. This because Java allows multiple methods with the same name, but different arguments.

- In the section where it was told the mechanism of exploring caller methods, it was stated that the tool is not capable of changing prototypes. In `WebGoat` there is a case that is not fixable because it needs the adding of some parameters to the method and for this reason the tool rises an exception. The code is reported below (**List of Codes 6.1**):

```java
protected AttackResult injectableQueryConfidentiality(String name
    , String auth_tan) {
    int index;
    ...
    String query = "SELECT * FROM employees WHERE last_name = ?
   AND auth_tan = ?";
    try (Connection connection = dataSource.getConnection()) {
        try {
            log(connection, query);
            PreparedStatement statement = connection.
    prepareStatement(query, ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
            index = 1;
            statement.setString(index++, name);
            statement.setString(index++, auth_tan);
            ResultSet results = statement.executeQuery();

            ...
        }
}

public static void log(Connection connection, String action) {
    ...
    String logQuery = "INSERT INTO access_log (time, action)
   VALUES ('" + time + "', '" + action + "')";
    try {
        statement.executeUpdate(logQuery, TYPE_SCROLL_SENSITIVE,
   CONCUR_UPDATABLE);
    }
}
```

List of Codes 6.1: Example of not fixable query.

As it can be observed in the method `log()`, we need to pass also some arguments to replace the placeholder inside `action` (`name` and `auth_tan`), but the tool is not capable of doing so.

6.4 Strange cases

It is crucial to mention some strange cases found in various projects. Indeed, since these are intentionally vulnerable applications, unconventional methods can be found in the code. In products of this kind, there are functions that cannot be corrected even manually. As an example, consider database queries where, in real cases, users can pass parameters that will later be inserted into the final query. However, in some method in `WebGoat`, the following is found:

```java
@PostMapping("/SqlInjection/attack4")
@ResponseBody
public AttackResult completed(@RequestParam String query) {
    return injectableQuery(query);
  }

protected AttackResult injectableQuery(String query) {
    ...
    statement.executeUpdate(query);
    ...
  }
```

List of Codes 6.2: Example of WebGoat not fixable.

As you can see, in this case, fixing it is impossible because the user is free to construct the query as they please. This means that this type of interaction cannot be addressed in any way. However, it is also important to consider that what we have seen will never be encountered in real cases where the interaction with the database follows predefined paths set by the programmer, so the user won't have this level of freedom. Below is the solution proposed by the tool:

```java
protected AttackResult injectableQuery(String query) {
    ...
    PreparedStatement statement_1 = connection.prepareStatement("?",
  TYPE_SCROLL_INSENSITIVE, CONCUR_READ_ONLY);
    index = 1;
    statement_1.setString(index++, query);
    statement_1.executeUpdate();
    ...
}
```

List of Codes 6.3: Proposed fix.

If we try to imagine that this is a normal case, it becomes evident that what is passed to the program is a `String`; therefore, the tool, in fixing this type of query, replaces the concatenated content with a placeholder “?”. In general, this case can be used to study the type of process being performed: the parameter is searched for in the code, its source is identified (in this example, it’s user input), and for this reason, it cannot be inserted directly but must be handled with a `PreparedStatement`.

In conclusion, it becomes evident that the proposed fix cannot be considered incorrect because it deals with an unusual case that the tool handles in a conventional manner. If this were indeed a typical case, then the proposed solution would be correct. This confirms the validity of the results just obtained and the resolution capability.

6.5 Not exploitable vulnerabilities

Some vulnerabilities found by SonarQube turned out to be **non-exploitable**. In fact, the input report theoretically should be filtered to remove this type of vulnerability. However, for testing purposes, these cases were also studied to verify their behavior. Before proceeding, it is necessary to distinguish between two scenarios:

1. The vulnerability is not exploitable, but if the code were to change in the future and this vulnerability were exposed to the user, it would become a true positive.

2. The vulnerability is a false positive from every perspective. This means that Sonar flagged it, but in reality, it is not exploitable and not even a vulnerability.

It is clear that among the two scenarios, the first one is more concerning, and it would be better if the tool were capable of correcting these cases as well. As mentioned in the previous sections, there are 18 vulnerabilities that belong to these two categories: 10 can be attributed to the first case, while the remaining 8 fall into the second case. Analyzing the results obtained, the tool performs flawlessly. It successfully corrects all 10 cases of the first type, while it encounters an error in fixing the other 8 of the second type. In reality, the fact that an exception is thrown for those 8 cases is not problematic because it means the tool has detected an anomaly and thus interrupts without modifying the code. Clearly, it cannot understand that the query is not vulnerable, but when the user reviews the code, this will be evident. Furthermore, the thrown exception is of type `IllegalStateException` so it will be easier to identify these cases. Below is a pie chart showing the corrections made:

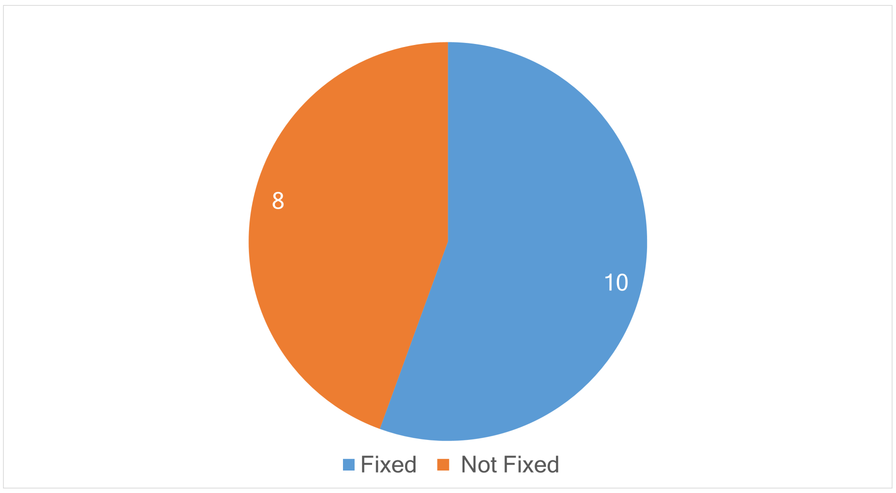


Figure 6.4: Pie chart of not exploitable vulnerabilities.

To avoid any misunderstandings and provide clarity, an example is provided for both types, and in the first case, the solution proposed by the tool will also be included.

```
    stmt.executeUpdate("CREATE DATABASE "+dbname);
```

List of Codes 6.4: Example of first type.

```
    stmt_1 = con.prepareStatement("CREATE DATABASE ?");
    index = 1;
    stmt_1.setString(index++, dbname);
```

List of Codes 6.5: Example of fix first type.

```
private static boolean isValidLogin(String query, String username,
    String password) {
    ...
        Connection connection = getConnection();
        PreparedStatement preparedStatement = connection.
    prepareStatement(query);
        preparedStatement.setString(1, username);
        preparedStatement.setString(2, password);
        ResultSet resultSet = preparedStatement.executeQuery();
    ...
}

public static boolean isValidUser(String username, String password) {
    String query = "SELECT COUNT(*) FROM PEOPLE WHERE ROLE = 'user'
   AND USER_ID = ? AND PASSWORD = ?";
    return isValidLogin(query, username, password);
}

public static boolean isValidAdmin(String adminUsername, String
    adminPassword) {
    String query = "SELECT COUNT(*) FROM PEOPLE WHERE ROLE = 'admin'
   AND USER_ID = ? AND PASSWORD = ?";
    return isValidLogin(query, adminUsername, adminPassword);
}

public static boolean isValidApiUser(String username, String password
    ) {
    String query = "SELECT COUNT(*) FROM PEOPLE WHERE USER_ID = ? AND
    PASSWORD = ?";
    return isValidLogin(query, username, password);
}
```

List of Codes 6.6: Example of second type.

As it can be seen from the example above, the code is already secure and not vulnerable, but SonarQube reported it as a SQL Injection. In this case the tool will throw an exception leaving the code untouched and in the final report, as result, there will be an explanation of the error, as mentioned before: "Illegal State".

6.6 Example of testing

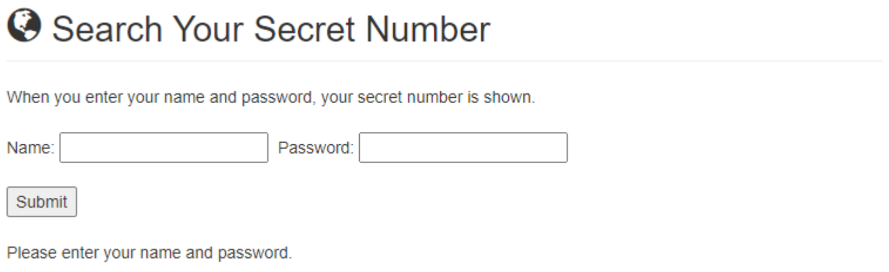
In this section, we will analyze an example of testing on the `easybuggy` project. Everything happens after configuring SonarQube with the FindBugs plugin and generating an auth_token.

Initially, the project was built so that SonarQube could analyze it. To do this, the following commands were executed:

- **mvn clean install**: this command is used to clean and rebuild the project from scratch;

- **mvn sonar:sonar -D"sonar.login=`auth_token`"**: this command triggers the analysis by SonarQube.

Once this phase is completed, the report is downloaded, and when running the project, various payloads are tested to exploit vulnerabilities. In the case of `easybuggy`, by going to the following url "`http://localhost:8080/sqlijc`" it is possible to insert username and password in order to get a personal secret of that specific user.


Search Your Secret Number

When you enter your name and password, your secret number is shown.

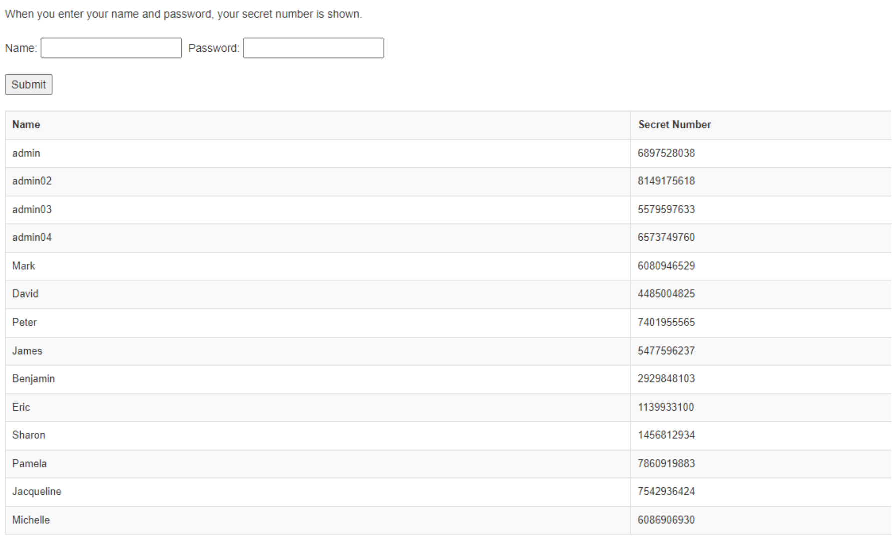
Name: Password:

Submit

Please enter your name and password.

Figure 6.5: EasyBuggy SQLInjection Home page.

Entering anything as the username and “’ OR ’1’=’1” as the password, all the secrets of all users can be obtained. This is, of course, a critical issue that needs to be resolved.


When you enter your name and password, your secret number is shown.

Name: Password:

Submit

Name	Secret Number
admin	6897528038
admin02	8149175618
admin03	5579597633
admin04	6573749760
Mark	6080946529
David	4485004825
Peter	7401955565
James	5477596237
Benjamin	2929848103
Eric	1139933100
Sharon	1456812934
Pamela	7860919883
Jacqueline	7542936424
Michelle	6086906930

Figure 6.6: EasyBuggy SQLInjection exploit.

After testing all the results from SonarQube, the report is inputted into the tool. Once various fixes have been applied (either locally or online), the code is rebuilt, and SonarQube is rerun using the same commands as before. By downloading the new report, it is verified that all vulnerabilities are no longer present. Therefore, the application is relaunched, and the same payload is sent to the same url. This time, since the fixed version is in use, the data of other users is no longer displayed, so the vulnerability is fixed.

6.7 Example results

In summary, the results obtained have been truly excellent. In fact, the corrected code is not only no longer vulnerable but also maintains the original logic. Furthermore, it's noticeable how the proposed solution tries to emulate the developer's programming style. This way, it will be more readable for the original developer.

6.7.1 Default statement simple cases

In this paragraph, some examples of completely vulnerable queries taken from various projects, such as `easybuggy`, `SQLInjectionTest` and `NotSecureBank`, will be inserted. The first one is the most simple case:

```
stmt = conn.createStatement();
rs = stmt.executeQuery("SELECT name, secret FROM users WHERE ispublic
    = 'true' AND name='" + name + "' AND password='" + password + "'"
    );
```

List of Codes 6.7: Default query, simple case from easybuggy.

The proposed solution is the following one:

```
PreparedStatement stmt = conn.prepareStatement("SELECT name, secret
    FROM users WHERE ispublic = 'true' AND name=? AND password=?");
index = 1;
stmt.setString(index++, name);
stmt.setString(index++, password);
rs = stmt.executeQuery();
```

List of Codes 6.8: Default query, fix simple case from easybuggy.

Now let's move into something a little more complex, instead of concatenating each parameters using "'", it is used "\'":

```
String sql = "SELECT * FROM sqlinjectiontest where name = " + "\'" +
    name + "\'";
ResultSet rs = stmt.executeQuery(sql);
```

List of Codes 6.9: Default query, case with "\'" from SQLInjectionTest.

In this case the tool is capable of solving the issue without any trouble:

```
String sql = "SELECT * FROM sqlinjectiontest where name = ?";
PreparedStatement stmt = conn.prepareStatement(sql);
index = 1;
stmt.setString(index++, name);
ResultSet rs = stmt.executeQuery();
```

List of Codes 6.10: Default query, fix case with "\'" from SQLInjectionTest.

It is clear that in simpler cases like the ones seen previously, the tool is perfectly capable of generating a solution. It can be noticed that, regardless of complexity, the `index` parameter is always added because incrementing it each time makes the configuration of each parameter simpler and safer.

6.7.2 Default statement complex cases

Now we will study progressively more complex cases. But before proceeding, let's define what the difficulties can be:

1. Parameters to add to initialization.
2. Bringing parts into the Setting phase.
3. Presence of multiple branches, e.g., if-then-else, for, while, etc.

These cases are clearly not mutually exclusive, which means that in the code, these difficulties could be combined. For example, you might have additional parameters to put in the initialization phase and also multiple branches (loops) where various variables are concatenated. In reality, all three cases could be present.

Starting with the first type, the following example is provided from `NotSecureBank`.

```
Statement statement = connection.createStatement();
statement.execute("INSERT INTO FEEDBACK (NAME,EMAIL,SUBJECT,COMMENTS)
    VALUES ('" + name + "', '" + email + "', '" + subject + "', '" +
    comments + "')", Statement.RETURN_GENERATED_KEYS);
ResultSet rs = statement.getGeneratedKeys();
```

List of Codes 6.11: Default query, multi parameteres initialization case from NotSecureBank.

As it can be seen, here the generated `ID` of the feedback is what is wanted, for this reason inside the initialization of `statement` it is present the field: `Statement-.RETURN_GENERATED_KEYS`. Whenever using a `PreparedStatement` this part must be brought inside the creation, so where it is involved the `connection` variable. Here it is reported the generated solution:

```
PreparedStatement statement = connection.prepareStatement("INSERT
    INTO FEEDBACK (NAME,EMAIL,SUBJECT,COMMENTS) VALUES (?, ?, ?, ?)",
    Statement.RETURN_GENERATED_KEYS);
index = 1;
statement.setString(index++, name);
statement.setString(index++, email);
statement.setString(index++, subject);
statement.setString(index++, comments);
statement.execute();
ResultSet rs = statement.getGeneratedKeys();
```

List of Codes 6.12: Default query, fix multi parameteres initialization case from NotSecureBank.

As for the second case, it can be thought of the SQL keyword "**LIKE**". Often, it is accompanied by the "%" symbol before or after the desired word. This is used for partial word matching, making the database search more powerful. Below is an example:

```
String query = "SELECT * FROM access_log WHERE action LIKE '%" +
    action + "%'";
Statement statement = connection.createStatement(ResultSet.
    TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet results = statement.executeQuery(query);
```

List of Codes 6.13: Default query, parts into setting phase case from WebGoat.

Actually, this case involves both type 1 and 2. Anyway, the tool generates a solution in which the logic is preserved and the vulnerability is no longer present in the code:

```
String query = "SELECT * FROM access_log WHERE action LIKE ?";
PreparedStatement statement = connection.prepareStatement(query,
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
index = 1;
statement.setString(index++, "%" + action + "%");
ResultSet results = statement.executeQuery();
```

List of Codes 6.14: Default query, fix parts into setting phase case from WebGoat.

Finally, to understand the third case, an example from `NotSecureBank` is provided, which is actually a mix of types 2 and 3. A parameter is concatenated using a "for" loop, while the `date` is initialized using an "if-then-else":

```
acctIds.append("ACCOUNTID = " + accounts[0].getAccountId());
for (int i = 1; i < accounts.length; i++) {
acctIds.append(" OR ACCOUNTID = " + accounts[i].getAccountId());
}
String dateString = null;
if (startDate != null && startDate.length() > 0 && endDate != null &&
     endDate.length() > 0) {
dateString = "DATE BETWEEN '" + startDate + " 00:00:00' AND '" +
    endDate + " 23:59:59'";
} else if (startDate != null && startDate.length() > 0) {
dateString = "DATE > '" + startDate + " 00:00:00'";
} else if (endDate != null && endDate.length() > 0) {
dateString = "DATE < '" + endDate + " 23:59:59'";
}
```

```java
String query = "SELECT * FROM TRANSACTIONS WHERE (" + acctIds.
    toString() + ") " + ((dateString == null) ? "" : "AND (" +
    dateString + ") ") + "ORDER BY DATE DESC";
resultSet = statement.executeQuery(query);
```

List of Codes 6.15: Default query, multi branch case from WebGoat.

The tool analyzes the given code and tries to generate a solution that maintains the programmer's style. Therefore, all the branches present are also copied to the Setting phase. The solution is shown below:

```java
acctIds.append("ACCOUNTID = ?");
for (int i = 1; i < accounts.length; i++) {
    acctIds.append(" OR ACCOUNTID = ?");
}
String dateString = null;
if (startDate != null && startDate.length() > 0 && endDate != null &&
     endDate.length() > 0) {
    dateString = "DATE BETWEEN ? AND ?";
} else if (startDate != null && startDate.length() > 0) {
    dateString = "DATE > ?";
} else if (endDate != null && endDate.length() > 0) {
    dateString = "DATE < ?";
}
String query = "SELECT * FROM TRANSACTIONS WHERE (" + acctIds.
    toString() + ") " + ((dateString == null) ? "" : "AND (" +
    dateString + ") ") + "ORDER BY DATE DESC";
ResultSet resultSet = null;
PreparedStatement statement = connection.prepareStatement(query);
if (rowCount > 0)
    statement.setMaxRows(rowCount);
index = 1;
statement.setLong(index++, accounts[0].getAccountId());
for (int i = 1; i < accounts.length; i++) {
    statement.setLong(index++, accounts[i].getAccountId());
}
if (!((dateString == null))) {
    if (startDate != null && startDate.length() > 0 && endDate !=
    null && endDate.length() > 0) {
        statement.setString(index++, startDate + " 00:00:00");
        statement.setString(index++, endDate + " 23:59:59");
    } else if (startDate != null && startDate.length() > 0) {
        statement.setString(index++, startDate + " 00:00:00");
    } else if (endDate != null && endDate.length() > 0) {
        statement.setString(index++, endDate + " 23:59:59");
    }
}
resultSet = statement.executeQuery();
```

List of Codes 6.16: Default query, fix multi branch case from WebGoat.

6.7.3 Partial statement

Before concluding, it is also needed to analyze partially vulnerable statements, meaning cases where there is a `PreparedStatement` with some parameters already inserted while others still need to be added. To do this, the tool must reconstruct the parameters that have already been inserted to understand where to insert the new ones. Below is a case from `WebGoat` and its corresponding solution.

```
String queryString = "SELECT * From user_data WHERE Login_Count = ?
    and userid= " + accountName;
PreparedStatement query = connection.prepareStatement(queryString,
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY)
query.setInt(1, count);
ResultSet results = query.executeQuery();
```

List of Codes 6.17: partial query from WebGoat.

In this case, the `count` is correctly concatenated but the `userid` is not concatenated using a placeholder. Looking inside the code, it is clear that the former must be replaced with a "?" and then it must be inserted in second position. The tool is capable of doing so by analysing where are all the original placeholders. So at the end, the obtained fix is the following one:

```
String queryString = "SELECT * From user_data WHERE Login_Count = ?
    and userid= ?";
PreparedStatement query = connection.prepareStatement(queryString,
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
index = 1;
query.setInt(index++, count);
query.setString(index++, accountName);
ResultSet results = query.executeQuery();
```

List of Codes 6.18: partial query fix from WebGoat.

Chapter 7

Conclusion

After analyzing the state of the art, it became evident that there was no existing product capable of automatically fixing code vulnerabilities. Clearly, the developed tool should not be considered a complete and production-ready solution; however, it serves as a starting point that marks a shift from the traditional approach where human intervention is indispensable. In fact, the tool could be integrated into the git workflow, making the analysis and resolution of vulnerabilities even simpler.

7.1 Pros of the tool

Compared to what already exists, this solution is **deterministic**, which means that if the input remains unchanged, the output will always be the same. This type of behavior is much harder to replicate when working with AI, which, by definition, generates results that are only probabilistically correct.

Furthermore, the tool **reduces human interaction**. In enterprise solutions, if there were vulnerabilities, they would traditionally need to be manually corrected. For instance, if there were 100 SQL Injections, a human operator would have to resolve each one manually. During the correction process, they might introduce new errors due to fatigue and/or distractions. By using the tool, all of this becomes impossible because human work is limited to reviewing the proposed solutions.

Moreover, since it's Java code, the entire project remains local without the need to upload files online. This means there is no reason to upload files to OpenAI or other companies. Integration with git will, of course, upload the code online; however, it is a tool for managing project versioning. If the developer deems it necessary, they can still perform the analysis locally. This is better for reason related to **privacy** and GDPR.

In addition to this, the use of the solution developed in this thesis is **more efficient** compared to using AI, for the same reason mentioned earlier; there is no need to upload any final information (except for the output on git).

All of these features allow a developer to **save time** on `vulnerability assessment`, allowing them to focus on other tasks that require greater attention. Consequently, the company will also become **more productive**.

As seen from the tests conducted, there are cases where the tool cannot resolve the vulnerability, but these are a small percentage. Therefore, the code review (which is still necessary) can be further expedited.

7.2 Cons of the tool

On the other hand, by **only addressing SQL Injection in Java**, the tool cannot be considered complete. Therefore, it currently needs to be supplemented with other solutions to achieve comprehensive coverage (such as making requests to ChatGPT via a web browser).

Furthermore, all the project on which the tool was tested were voluntary vulnerable, this means that they cannot be compared with a real application, as it was explained in the previous chapter (**Section 6.4**).

7.3 Future works

Firstly, it must be highlighted, that in the future the tool can be tested on real-world applications, this to achieve more reliability and credibility. Indeed, this step is fundamental before being adopted in enterprise solutions.

Furthermore, in the future, the project can be extended. In fact, the code has been designed in a **modular way**, as explained earlier; the `Vulnerability` class can be implemented and extended by many other classes in the future. This work will allow for the management of many more vulnerabilities. Additionally, the "JavaParserUtil" Java file is already present, with many useful methods for code parsing, to expedite further development.

Certainly, among the first ideas is the **addition of various vulnerabilities** to fix, but there is also a desire to work on **multiple languages**. Starting with the most interesting vulnerabilities, there are various Injections, such as XSS and Command Injection, but other categories in the OWASP top 10 should not be excluded.

As for the languages to address, JavaScript undoubtedly takes the top spot, being one of the most widely used languages in the web domain, followed by Python and PHP. Of course, the aim is to follow market trends, trying to modernize and make the tool more complete, in the hope that one day it will be capable of fixing vulnerabilities in every programming language.

Bibliography

[1] "OWASP A01 Broken Access Control", [Online] Available: https://owasp.org/Top10/A01_2021-Broken_Access_Control/

[2] "A02:2021 – Cryptographic Failures", [Online] Available: https://owasp.org/Top10/A02_2021-Cryptographic_Failures/

[3] "A03:2021 – Injection", [Online] Available: https://owasp.org/Top10/A03_2021-Injection/

[4] "A04:2021 – Insecure Design", [Online] Available: https://owasp.org/Top10/A04_2021-Insecure_Design/

[5] "A05:2021 – Security Misconfiguration", [Online] Available: https://owasp.org/Top10/A05_2021-Security_Misconfiguration/

[6] "A06:2021 – Vulnerable and Outdated Components", [Online] Available: https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

[7] "A07:2021 – Identification and Authentication Failures", [Online] Available: https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/

[8] "A08:2021 – Software and Data Integrity Failures", [Online] Available: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/

[9] "A09:2021 – Security Logging and Monitoring Failures", [Online] Available: https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/

[10] "A10:2021 – Server-Side Request Forgery (SSRF)", [Online] Available: https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/

[11] "OWASP Top 10:2021", [Online] Available: https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/ "OWASP Top 10:2021", [Online] Available: https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/

[12] A. Sadeghian, M. Zamani and A. A. Manaf, "A Taxonomy of SQL Injection Detection and Prevention Techniques", 2013 International Conference on

Informatics and Creative Multimedia, Kuala Lumpur, Malaysia, 2013,pp. 53-56, DOI: 10.1109/ICICM.2013.18.

[13] W. G. J. Halfond. A Classification of SQL Injection Attacks and Countermeasures. N.p., 2006. Print.

[14] Y. Pan, “Interactive Application Security Testing”, 2019 International Conference on Smart Grid and Electrical Automation (ICSGEA), Xiangtan, China, 2019, pp. 558-561, DOI: 10.1109/ICSGEA.2019.00131.

[15] “SonarQube 10.2 Documentation”, [Online] Available: `https://docs.sonarsource.com/sonarqube/latest/`

[16] “Quality profiles”, [Online] Available: `https://docs.sonarsource.com/sonarqube/latest/instance-administration/quality-profiles/`

[17] “Find Security Bugs”, [Online] Available: `https://find-sec-bugs.github.io/`

[18] “Potential SQL Injection”, [Online] Available: `https://find-sec-bugs.github.io/bugs.htm#SQL_INJECTION`

[19] “Application Security Testing Reviews and Ratings”, [Online] Available: `https://www.gartner.com/reviews/market/application-security-testing`

[20] “Mend SAST”, [Online] Available: `https://www.mend.io/sast/`

[21] “Peerspot, Mend.io Reviews”, Jeffrey Harker, 2022, [Online] Available: `https://www.peerspot.com/products/mend-io-reviews#review_2284212`

[22] “Gartner Mend.io Reviews”, 2022, [Online] Available: `https://www.gartner.com/reviews/market/application-security-testing/vendor/mend-io/product/mend/review/view/4533780`

[23] “Gartner Mend.io Reviews”, 2022, [Online] Available: `https://www.gartner.com/reviews/market/application-security-testing/vendor/mend-io/product/mend/review/view/4534128`

[24] “Veracode Fix”, [Online] Available: `https://www.veracode.com/fix`

[25] “CheckMarx SAST”, [Online] Available: `https://checkmarx.com/cxsast-source-code-scanning/`

[26] “CheckMarx SAST”, [Online] Available: `https://about.gitlab.com/topics/devops/`

[27] “Snyk Code”, [Online] Available: `https://snyk.io/product/snyk-code/`

[28] “Sourcegraph”, [Online] Available: `https://sourcegraph.com/search`

[29] “Is ChatGPT the Ultimate Programming Assistant – How far is it?”, Haoye Tian and Weiqi Lu and Tsz On Li and Xunzhu Tang and Shing-Chi Cheung and Jacques Klein and Tegawendé F. Bissyandé, 2023, [Online] DOI: 10.48550/arXiv.2304.11938

[30] “gpt3_security_vulnerability_scanner”, chris-koch-penn, [Online] `https://github.com/chris-koch-penn/gpt3_security_vulnerability_scanner`

[31] "I Used GPT-3 to Find 213 Security Vulnerabilities in a Single Codebase", Chris Koch, 2023 [Online] https://betterprogramming.pub/i-used-gpt-3-to-find-213-security-vulnerabilities-in-a-single-codebase-cc387

[32] "CodeGPT", [Online] https://plugins.jetbrains.com/plugin/21056-codegpt

[33] "GPTCoder", [Online] https://marketplace.visualstudio.com/items?itemName=codista.vscodewriter

[34] "JavaParser", [Online] https://javaparser.org/

[35] "JavaParser About", [Online] https://javaparser.org/about.html

[36] "JavaParser Documentation: class CompilationUnit", [Online] https://javadoc.io/static/com.github.javaparser/javaparser-core/3.2.8/com/github/javaparser/ast/CompilationUnit.html

[37] "JavaParser Documentation: class Node", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/Node.html

[38] "JavaParser Documentation: class Range", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/Range.html

[39] "JavaParser Documentation: class MethodDeclaration", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/body/MethodDeclaration.html

[40] "JavaParser Documentation: class Statement", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/body/MethodDeclaration.html

[41] "JavaParser Documentation: class BlockStmt", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/stmt/BlockStmt.html

[42] "JavaParser Documentation: class IfStmt", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/stmt/IfStmt.html

[43] "JavaParser Documentation: class WhileStmt", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/stmt/WhileStmt.html

[44] "JavaParser Documentation: class DoStmt", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/stmt/DoStmt.html

[45] "JavaParser Documentation: class ForStmt", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/stmt/ForStmt.html

[46] "JavaParser Documentation: class TryStmt", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/

3.25.5/com/github/javaparser/ast/stmt/TryStmt.html

[47] "JavaParser Documentation: class ExpressionStmt", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/stmt/ExpressionStmt.html

[48] "JavaParser Documentation: class Expression", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/expr/Expression.html

[49] "JavaParser Documentation: class AssignExpr", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/expr/AssignExpr.html

[50] "JavaParser Documentation: class NameExpr", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/expr/NameExpr.html

[51] "JavaParser Documentation: class VariableDeclarationExpr", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/expr/VariableDeclarationExpr.html

[52] "JavaParser Documentation: class MethodCallExpr", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/expr/MethodCallExpr.html

[53] "JavaParser Documentation: class ConditionalExpr", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/expr/ConditionalExpr.html

[54] "JavaParser Documentation: class StringLiteralExpr", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/ast/expr/StringLiteralExpr.html

[55] "JavaParser Documentation: interface TypeSolver", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-core/3.25.5/com/github/javaparser/resolution/TypeSolver.html

[56] "JavaParser Documentation: class CombinedTypeSolver", [Online] https://www.javadoc.io/static/com.github.javaparser/javaparser-symbol-solver-core/3.23.1/com/github/javaparser/symbolsolver/resolution/typesolvers/CombinedTypeSolver.html

[57] "JavaParser Documentation: class JavaSymbolSolver", [Online] https://www.javadoc.io/static/com.github.javaparser/java-symbol-solver-core/0.6.3/com/github/javaparser/symbolsolver/JavaSymbolSolver.html

[58] "JavaParser Documentation: class ReflectionTypeSolver", [Online] https://www.javadoc.io/static/com.github.javaparser/java-symbol-solver-core/0.6.3/com/github/javaparser/symbolsolver/resolution/typesolvers/ReflectionTypeSolver.html

[59] "Package one.util.streamex", [Online] `http://amaembo.github.io/streamex/javadoc/one/util/streamex/package-summary.html`
[60] "Stream methods: distinct", [Online] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#distinct--`
[61] "package org.json", [Online] `https://javadoc.io/doc/org.json/json/latest/index.html`
[62] "Maven Invoker Plugin", [Online] `https://maven.apache.org/plugins/maven-invoker-plugin`
[63] "Package org.gradle.tooling", [Online] `https://docs.gradle.org/current/javadoc/org/gradle/tooling/package-summary.html`
[64] "Apache Commons CLI", [Online] `https://commons.apache.org/proper/commons-cli`
[65] "Eclipse JGit™", [Online] `https://www.eclipse.org/jgit/`
[66] "GitHub API for Java", [Online] `https://github-api.kohsuke.org/`
[67] "HttpClient Overview", [Online] `https://hc.apache.org/httpcomponents-client-5.1.x/index.html`
[68] "NotSecureBank", [Online] `https://github.com/enzopalmisano/NotSecureBank`
[69] "easybuggy", [Online] `https://github.com/k-tamura/easybuggy`
[70] "java-sec-code", [Online] `https://github.com/JoyChou93/java-sec-code`
[71] "JavaSec-SQLInject-Demo", [Online] `https://github.com/UzJu/JavaSec-SQLInject-Demo`
[72] "JavaVulnerableLab", [Online] `https://github.com/CSPF-Founder/JavaVulnerableLab`
[73] "SQLInjectionTest", [Online] `https://github.com/AchillesCap/SQLInjectionTest`
[74] "tarpit-java", [Online] `https://github.com/ShiftLeftSecurity/tarpit-java`
[75] "vulnado", [Online] `https://github.com/ScaleSec/vulnado`
[76] "WebGoat", [Online] `https://github.com/WebGoat/WebGoat`