BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PLUGIN FOR A CODE RISK ANALYSIS

PLUGIN PRO ANALÝZU RIZIK V KÓDU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAROSLAV KVASNIČKA**
AUTOR PRÁCE

SUPERVISOR **Ing. VOJTĚCH HAVLENA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

146738

Institut:	Department of Intelligent Systems (UITS)
Student:	**Kvasnička Jaroslav**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Plugin for a Code Risk Analysis**
Category:	Software analysis and testing
Academic year:	2022/23

Assignment:

1. Study techniques of static analysis of code and methods for detecting possibly vulnerable behavior.
2. Get acquainted with a creation of plugins into Eclipse.
3. Develop suitable techniques allowing to check the code in Java/Javascript for various risky behavior (such as dangerous string values, obsolete code, etc.) with a particular focus on user accessibility.
4. Implement the proposed techniques as an Eclipse plugin and evaluate them on various scenarios.
5. Discuss the achieved results and further possible extensions.

Literature:

- RIVAL, Xavier a Kwangkeun YI. *Introduction to static analysis: an abstract interpretation perspective*. Cambridge: The MIT Press, 2020, xiv, 299 pages. ISBN 978-0-262-04341-0.
- D. Evans and D. Larochelle, *Improving security using extensible lightweight static analysis*, in *IEEE Software*, vol. 19, no. 1, pp. 42-51, Jan.-Feb. 2002, doi: 10.1109/52.976940.
- Holzner, S. *Eclipse Cookbook.* O'Reilly Media, 2004, 343 pages. ISBN 9780596007102.

Requirements for the semestral defence:
First **three** items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Havlena Vojtěch, Ing., Ph.D.**
Consultant:	Maria Ana Casal Cunha
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

This thesis explores the importance of code analysis and testing and its impact on program functionality and cost. It discusses the basic principles and types of testing, with a focus on static analysis. The Eclipse platform and plugins are also examined, along with the architecture of the target code. We then discussed and implemented an extension of the static MFB analyzer which the BOC-GROUP company developed along with extensions such as the Log4j checker and GDPR analyzer. We also designed and implemented the Eclipse plugin which can highlight the results of the MFB analyzer in the Eclipse IDE. In conclusion, future possible extensions are also discussed, including harmful regular expression detectors and XSS prevention.

Abstrakt

Tato práce se zabývá významem analýzy a testování kódu a jeho dopadem na funkčnost a náklady programu. Pojednává o základních principech a typech testování se zaměřením na statickou analýzu. Zkoumá také platformu Eclipse a zásuvné moduly a architekturu cílového kódu. Dále jsme probrali a implementovali rozšíření statického analyzátoru MFB, který byl vyvinut společností BOC-GROUP, spolu s rozšířeními, jako je kontrola Log4j a analyzátor GDPR. Navrhli jsme a implementovali také zásuvný modul pro Eclipse, který dokáže zvýraznit výsledky analyzátoru MFB v prostředí Eclipse IDE. V závěru jsou také diskutována možná budoucí rozšíření, včetně detektorů škodlivých regulárních výrazů a prevence XSS.

Keywords

Static Analysis, Eclipse, Plugin, Whitebox testing, False positive

Klíčová slova

Statická analýza, Eclipse, Plugin, testování bílé skříňky, falešně pozitivní

Reference

KVASNIČKA, Jaroslav. *Plugin for a Code Risk Analysis.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vojtěch Havlena, Ph.D.

Rozšířený abstrakt

V dnešním světě je software nezbytnou součástí prakticky všech aspektů našeho života. Používá se ve všech oblastech, od dopravních systémů přes finanční služby až po zdravotnictví. Zajištění kvality a bezpečnosti softwaru je proto stále důležitější, zejména vzhledem k vysokým nákladům spojeným s chybami a selháními softwaru. Důsledky chyb v softwaru mohou být závažné, od finančních ztrát až po poškození dobrého jména společnosti, a v některých případech dokonce ohrožení lidských životů. Společnosti zabývající se vývojem softwaru proto musí k testování a zajišťování kvality přistupovat proaktivně, aby k takovým incidentům nedocházelo.

Vývoj softwaru je složitý proces, který zahrnuje několik fází a týmů. Aby byla zajištěna kvalita softwaru, je nutné jej testovat v každé fázi vývoje. To zahrnuje nejen funkční testování, ale také testování nefunkčních požadavků, jako je bezpečnost a výkon. Jedním ze způsobů testování kvality a bezpečnosti softwaru je statická analýza, na kterou se v této práci budeme zaměřovat.

Statická analýza je proces analýzy kódu, aniž by byl software skutečně spuštěn. Zahrnuje použití specializovaných nástrojů, které zkoumají kód a hledají v něm běžné chyby, zranitelnosti a další problémy. Statická analýza může být obzvláště užitečná pro identifikaci chyb v rané fázi procesu vývoje, ještě předtím, než je software nasazen a může způsobit škody.

Jedním z problémů statické analýzy je problém falešně pozitivních výsledků. K falešně pozitivním výsledkům dochází, když nástroj analýzy identifikuje problém, který ve skutečnosti problémem není. To může vést k promarnění času a úsilí, protože vývojáři tráví čas zkoumáním problémů, které ve skutečnosti neexistují. Pro řešení tohoto problému je důležité používat nástroje, které jsou při identifikaci problémů přesné a účinné.

Při psaní této práce jsme spolupracovali s firmou BOC-GROUP, která vyvinula program ADONIS, který je široce využíván ve finančním sektoru. Program ADONIS je komplexní aplikace, která vyžaduje pečlivé testování a analýzu, aby byla zajištěna její kvalita a bezpečnost. Za tímto účelem společnost vyvinula MFB Analyzer, nástroj statické analýzy, který je speciálně určen k analýze programu ADONIS.

Cílem této bakalářské práce je podrobně prozkoumat využití nástrojů statické analýzy při vývoji softwaru, se zvláštním zaměřením na analyzátor MFB a jeho použití v systému ADONIS a jeho rozšířeních. Práce nabídne důkladné prozkoumání koncepce statické analýzy včetně jejích výhod a nevýhod a také představí reálné příklady analýzy založené na pravidlech, které ilustrují její praktické využití.

Kromě toho práce poskytne přehled o populárním integrovaném vývojovém prostředí Eclipse a jeho roli při podpoře statické analýzy. Na konci práce by čtenáři měli důkladně porozumět statické analýze, její úloze při vývoji softwaru a tomu, jak lze Eclipse využít k usnadnění procesu analýzy. Průzkum provedený v této práci poslouží jako cenný zdroj informací pro vývojáře softwaru i pro zájemce o oblast softwarového inženýrství.

Zaměříme se na architekturu programu ADONIS a nástroje MFB Analyzer a popíšeme si, jak se MFB Analyzer využívá pro statickou analýzu. Navrhneme a implementujeme

rozšíření programu MFB Analyzer, jako je kontrola Log4j a vyhledávač citlivých údajů v souladu s obecným nařízením o ochraně osobních údajů (GDPR). A budeme diskutovat výsledky a problémy kterým jsme čelili během vývoje.

Prozkoumáme platformu Eclipse a s ní spojené zásuvné moduly a budeme se zabývat různými nástroji statické analýzy dostupnými jako zásuvné moduly Eclipse, jejich vlastnostmi, silnými a slabými stránkami. Vyvineme a implementujeme také zásuvný modul pro značení v Eclipse, který umožní přímé zobrazení a zvýraznění výsledků kódu analyzátoru MFB Analyzer v prostředí Eclipse IDE, což povede ke zjednodušení procesu pro vývojáře. Z toho budou těžit vývojáři společnosti BOC-GROUP, kteří tento plugin používají v životním cyklu vývoje, což zefektivní jejich práci a sníží náchylnost k chybám. Opět si vše shrneme a popíšeme si jak výsledky práce tak překážky, kterým jsme čelili během vývoje.

V závěru práce budou diskutována potenciální budoucí rozšíření analyzátoru MFB, jako je detektor škodlivých regulárních výrazů, kontrola útoků XML bomb, prevence XSS prostřednictvím JSP a detekce útoků nezabezpečenými náhodnými funkcemi. Tato rozšíření by mohla dále zlepšit schopnosti analyzátoru MFB a zvýšit bezpečnost a kvalitu programu ADONIS.

Plugin for a Code Risk Analysis

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Vojtěch Havlena Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jaroslav Kvasnička
May 7, 2023

Acknowledgements

I would like to acknownledge my warmest thanks to my supervisor Vojtěch Havlena for leading this thesis, and to my technical supervisor Maria Casal Cunha for leading the practical part of this thesis.

Contents

List of Figures

Chapter 1

Introduction

This bachelor thesis will focus on the importance of code analysis, exploring how the failure to properly test programs in the past has led to disastrous consequences for the systems and the end-users. We will also examine the costs associated with repairing bugs in software, highlighting the importance of early detection and prevention of these issues. In addition, we will explore the different types of testing methods that are commonly used in the software development process and their respective advantages and disadvantages.

The thesis will provide a comprehensive overview of static analysis and will showcase real-world examples of rule-based analysis, explaining how the rules work and the importance of this technique in detecting errors in code.

False positives are a common issue that can arise during static analysis, and the thesis will outline strategies for avoiding these errors. Furthermore, we will explore the Eclipse platform and its associated plugins. We will examine various static analysis tools available as Eclipse plugins, their properties, strengths, and weaknesses. We will then dive into the architecture of the tested system and the tool that will be integrated into the Eclipse platform as a plugin.

We will dive into the architecture of the code targeted by the BOC-GROUP company and its extensions, and explore how the MFB Analyzer is utilized for static analysis. We will begin by providing an in-depth analysis of the MFB Analyzer's architecture, and describe how it operates. We will then design and implement MFB Analyzer extensions, such as the Log4j checker and the sensitive data seeker in accordance with the General Data Protection Regulation (GDPR). We will discuss the Log4j exploit in detail, highlighting why it is a significant threat. Furthermore, we will explore the GDPR's importance to both organizations and individuals and provide insights into how we tackled the challenges associated with preparing for and implementing these extensions. We will then design and implement the Eclipse marking Extension. This extension will allow the MFB Analyzer's code results to be directly shown and highlighted within the Eclipse IDE, resulting in a streamlined process for developers. We will describe the obstacles we encountered during the preparation and implementation phases of this extension and how we overcame them, we will also describe the benefits which will these extensions bring to the developers using them.

Finally, we will discuss more possible extensions that could be added to the MFB Analyzer in the future, such as a harmful regular expression detector, an XML bomb attack checker, an XSS via JSP prevention, and an insecure random functions attack detection.

Chapter 2

Analysis/Testing

Testing of code is an essential part of the software development life cycle. It is used for verification of the code. Verification is determining if the correct functionality is being developed and if the software is developed according to specified requirements. It should cover a large portion of the software development life cycle and show the developers, what functionality is not correct and where the problem is. For this purpose, many testing concepts have been created and are used daily all over the IT industry. The most common types are white-box testing, black-box testing, and grey-box (gray-box) testing.

2.1 Why Is Testing Important?

From historical events, we learn very important and costly lessons. Here are some examples of system failures due to bad programming.

2.1.1 Ariane 5

Ariane 5 was a rocket delivering 2 satellites belonging to the European telecom consortium and the French space research institute CNES to orbit. Unfortunately, after 37 seconds after the start, the rocket flipped 90° in the wrong direction and was ripped apart at a height of 4km. The resulting damage cost over $400 million.

After inspection, it was found that the cause of the failure of the rocket was a bug in its inertial reference system. There was a conversion for acceleration from a 64-bit number down to a 16-bit variable. When the speed of the rocket increased and the 64-bit variable increased over 65k, it resulted in an overflow of the 16-bit variable. which triggered operand mathematical calculation errors. The backup of the Inertial Reference System also failed due to the same error and by mistake defined it as actual flight data. [10]

2.1.2 Mars Climate Orbiter

It was a research satellite launched by NASA in 1998 to research the Mars atmosphere. After the satellite entered its atmosphere it lost communication with the Earth. The mission was declared a failure.

Later calculations have shown that the research probe flew over the Martian surface at a height of 57km instead of 150km, burning the probe.

The primary cause of the failure was a bug in the software. The first program was using the probe producing results in United States customary units while the second program

was using SI units. The software calculated thrust firing force in pound-force seconds, while it was expected to be in newton-seconds, incorrect by a factor of 4.45.

The resulting cost was $327,6 million at that time. [34]

2.1.3 Great Worm

Otherwise called Morris worm was one of the oldest worms distributed via the internet. It went from a harmless computing exercise to a mass denial of service attack, infecting thousands of computers. The worm was able to copy itself on the same machine, slowing it down a causing it to crash. The total economic impact was between $10 000 to $10 000 000. [35]

2.2 Principles Of Testing

This part was inspired by the Software Testing – Goals, Principles, and Limitations. More information about the principles of testing can be found in this reference. [23]

- Setting correct goals is very important. At first, we need to determine what we want to achieve in the testing, what we test, and what are the expected results.
- Start testing early in the development. The earlier error is caught, the lower the price and time for its repair.

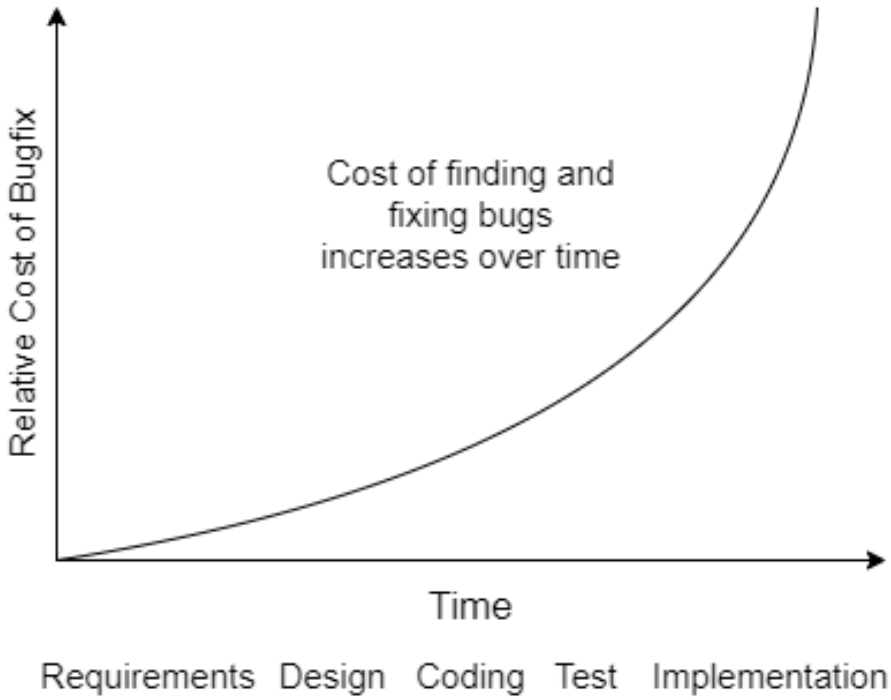


Figure 2.1: Cost of fixing bugs in different phases [23]

- Set effective test cases. If the tester lacks an understanding of the architecture and the requirements of the program, then he surely can not design efficient tests.
- Use both static and dynamic testing.

2.3 BlackBox Testing

This type of testing is performed when the code has been compiled and its functionality can be tested in the runtime. With the black-box approach, the tester does not need to know the code to perform the tests. Since the testing tools have only the knowledge of the inputs and outputs of the program, they may find bugs that wouldn't be otherwise found with white-box testing. The disadvantages of black-box testing are, that it can not be used in the early development stage, since it tests running applications and is limited to the database of the testing tools. One of the advantages is that it has fewer false positives than statical analysis. Some of the tools used for these purposes are for example Selenium, AutoHotKey, or AegisWeb. [22]

2.4 GrayBox Testing

Gray-box testing is partially based on black-box testing and partially on white-box testing. The person testing the software must have at least partial knowledge of the program's internal composition. It differs from black-box testing in the depth of searching for information. For example in black-box testing the tool checks if a button was clicked and then if a website opened. During the gray-box testing, the tool checks if the button was clicked, what was the request sent and received, checks for the IDs of the user and other details of the message, and then verify, if the website opened. Some of the tools used for gray-testing are Selenium, Appium, and more. [22] [26]

2.5 WhiteBox Testing

Also known as clear box or glass box testing, it is a type of testing where the tester has full knowledge of the tested code. It is based on testing structures and internal parts of the program, rather than its functionality. Whitebox testing has the ability to show the developer, which line of code has been executed and therefore points to a problem directly in the code, that's why many white-box type testing tools are direct integrations in IDE's. [22] [27]

2.5.1 Unit Testing

The code is tested by its smallest constructions, to determine if basic parts and modules have the correct functionality. Tests are usually prepared by developers although mostly executed automatically with predefined test cases. Unit testing comes in handy especially when code is being refactored, then the developer will know, that the refactored module has kept its correct functionality. There are also many other techniques of testing for example dynamic testing, code coverage, mutation testing, and more. But it has also disadvantages, one of them being that it can be used for small parts of a system. [27]

2.5.2 Static Application Security Testing

Static application security testing or SAST in short is a technique of white-box testing focused on security vulnerabilities. For example detecting SQL injections, secure cookie transmission failure, generation of weak cryptographic material and passwords, or uninitialized variables. Its functionality is to scan through the code and pinpoint flaws and

vulnerabilities by the predefined rules of different severity categories. Defined rules are explained at the beginning of the chapter on static analysis. At first, static application security testing will detect a high number of warnings most of which will be false positives, because the tool using this technique does not consider the whole program. After being set up it can detect very well any other vulnerable code and it can categorize it by different levels of severity. For the security analysis to be complete, the static application security testing should be followed by dynamic application security testing and if needed, then interactive application security testing.

Chapter 3

Static Analysis

3.1 Introduction

Static analysis is a technique used in software development to evaluate the quality of code without actually executing it. It involves analyzing the source code, either manually or using automated tools, to identify potential errors, security vulnerabilities, and other issues before the code is run.

Static analysis tools typically scan the source code and compare it against a set of predefined rules or patterns. These rules can include expressions, structures, or simple strings that the tool is programmed to look for. For example, a tool might search for common coding mistakes such as uninitialized variables, buffer overflows, or null pointer dereferences. It may also check for adherence to coding standards and best practices, such as proper formatting, naming conventions, and documentation.

It is strongly dependent on the programming language used in the development process. Each language has its own unique syntax, semantics, and potential pitfalls, so static analysis tools must be designed to work with specific languages. For example, a tool that works well for analyzing Java code may not be as effective when applied to Python code.

One advantage of static analysis is that it can be performed early in the development cycle before the code is compiled or deployed. This allows developers to catch errors before they cause problems downstream, saving time and resources.

It is not a substitute for thorough testing, and it cannot detect all types of errors or security vulnerabilities. Moreover, static analysis tools can generate false positives or false negatives, leading to unnecessary work or missed issues.

3.2 Rules Based Analysis

Predefined rules in the static analysis are a set of guidelines, patterns, or conditions that are defined in advance to check the source code for potential issues. These rules are based on best practices, coding standards, and common programming mistakes, and are used by static analysis tools to identify potential problems in the code.

Predefined rules can take many forms, such as regular expressions, syntax patterns, or code snippets, depending on the type of issue being checked. One advantage of predefined rules in static analysis is that they allow developers to catch potential issues early in the development cycle. By using automated tools to check for common programming mistakes

and adherence to coding standards, developers can save time and effort and improve code quality.

Here are a couple of examples of different rules:

- **UninitializedVariable** Checks all variables and marks uninitialized variables

```
For each variable declaration:
    If the variable is not assigned a value at the point of declaration,
        mark it as uninitialized.
For each usage of a variable:
    If the variable is used before it is assigned a value,
        mark it as uninitialized.
```

- **SqlQuery** Looks for any SQL statements in the code using regular expression

```
(SELECT|INSERT|UPDATE|DELETE|CREATE|ALTER|DROP|TRUNCATE)[^;]*
```

- **UnusedImports** Looks for any unused imports. The rule is declaratively defined

```
Rule: UnusedImports
Description: Check for unused imports in Java code
Priority: High
When:
  - The Java source code is compiled
Then:
  - Flag any import statements that are not used in the code
```

- **TooLongVariableName** Checks variables length

```
List<String> ListVariables = getAllVariables()
for(String variable : ListVariables)
    if(variable.length >= 30)
        MarkVariable(line, variable, file)
```

Here is an example of tested source code:

```
void function() {
    int x;
    int y = 10;
    int z;
    if (y > 0)
        x = 5;
}
int main() {
    function();
    List<String> usersVariableWithAReallyLongName =
        db.connect().sql("SELECT * FROM User");
    return 0;
}
```

At this moment if we would run our static analyzer with the rules mentioned above we would get these results and these rows would be marked as positive findings.

```
void function() {
    int x;
    int y = 10;
    int z;                                          //UninitializedVariable
    if (y > 0)
        x = 5;
}
int main() {
    function();
    List<String> usersVariableWithAReallyLongName = //TooLongVariableName
        db.connect().sql("SELECT * FROM User");     //SqlQuery
    return 0;
}
```

Finding Report:

- Rule: UninitializedVariable
  - line: 3
  - variable: z
- Rule: TooLongVariableName
  - line: 9
  - variable: usersVariableWithAReallyLongName
  - function: main
- Rule: SqlQuery
  - line: 10
  - SQL: SELECT * FROM User

We utilized a static code analyzer to evaluate the source code and identify potential issues based on predefined rules. The analyzer parsed the code and applied each rule separately to identify different types of problems.

For instance, the first rule aimed to detect uninitialized variables by keeping a list of all variables in the code and removing those that were initialized. At the end of parsing, the rule reported any variable that was still in the list as uninitialized.

The second rule looked for long variable names, where all variables in the code were checked if their length exceeded 30 characters. If a variable's name was too long, it would be marked as a problem.

The third rule utilized a regular expression to identify any SQL statements present in the code. When the analyzer found any matches, it would report them as a potential issue.

Once the static code analyzer completed its analysis, it printed a report summarizing the findings. The report included details such as the rule that triggered the problem, the line number where the issue was detected, and any other relevant information.

Here is a second example using rules from the PMD tool:

The rule can be called for example UnusedPrivateMethod. We can have this example in the pseudo-code [19]:

```
rule UnusedPrivateMethod{
    for(Method method : ListAllMethods){
        Method mainMethods = getMainMethods()
        if(!mainMethods.contains(method))
            UnusedMethodFound(line, method, file)
    }
}
```

In this code example, we have a rule called UnusedPrivateMethod which is applied to a source code. There is a ListAllMethods which contains all methods used in our source code. The function getMainMethods will get us only methods used in the main function and for comparison, there is a condition statement that checks if all of the methods are used in the main function, if there are some methods missing, the rules mark it as a positive finding.

In this source code example, we have two classes each containing private functions. Only one of the functions is called in the main function. We will apply the rule Unused-PrivateMethod on this source code.

```
    private class Someting{
        private void foo(){}  //rule UnusedPrivateMethod would be triggered
    }
    private class Used{
        private void fcc(){}
    }
    void main(){
        User.fcc()
    }
```

The Rule called Unused private method is defined in a Java class called UnusedPrivateMethodRule.java referenced in bibliography [18]. In the source code, it will retrieve the occurrences from a list of all methods by the name of the method. Called in this example method "foo" and "fcc". If the method has no occurrence in the source code, it will be marked as a

positive finding and will be added as a violation to the report, thus the rule had fired. Else it will check if the method has not been called from any other method, if not, the rule will add a positive finding and add a violation to the report. Here is part of the code from the UnusedPrivateMethodRule.java class line 84:

```
List<NameOccurence> occs = methods.get(methodName);
if(occs.isEmpty()){
    addViolation(data, mnd.getNode(),
        mnd.getImage() + mnd.getParameterDisplaySignature());
} else {
if(isMethodNotCalledFromOtherMethods(mnd, occs)){
    addViolation(data, mnd.getNode(),
        mnd.getImage() + mnd.getParameterDisplaySignature()); }}
```

After this rule has been applied to the source code, we could have this summary report:

Finding Report

- Searched methods: 2
  - foo
  - fcc
- Positive occurrences: 1
  - foo
    - expl: Violated UnusedPrivateMethodRule
    - src: CodeExample/example.code:2

When the analysis tool goes over this particular part of the source code, it fires, because it knows that the method „foo“ in this example is not used anywhere else. If a rule fires, then it is known that there is an error in the code, but that can cause many false positives because the analysis does not take the whole picture of the program into account.

3.3 False Positive Reports Of Static Analysis

The tools that perform static analysis are not perfect and therefore they will always result in some false positive calls. False positives are called incorrect reports of static analyzers. As code is getting more complex, static analyzing tools run into code that executes some rules which marks it as a positive finding. We need to get rid of false positive findings to perform efficient analysis. If analysis produces many false positive findings, it is time-consuming and frustrating for the users to work with such a tool. One of the methods to remove false positives from the analysis is to make more complex rules, that will be able to detect complex code structures which are the product of false positive calls, another method is to have a list of exceptions for each false positive finding. This list can be created from the reports by the user by marking false positive findings, which will not be listed in the reports in the next run of the analyzing tool. Here are some example sources inspired by the CodeChecker static analysis tool, since most of the static analysis tools work similarly, differencing only in details of their implementation, we can anticipate the code to behave the same way, as it would in different static analysis tools. [4]

For example, we have a very simple rule shown in the pseudo-code below. The rule should open a source code file, read all strings and split them into words, and save them in a buffer, which will be looped over and checked for string „evaluation“. If there will be an occurrence of the string „evaluation“, it will be added to the report as a positive finding.

```
FILE f = new FILE("src.code")
string buffer[] = f.readWords()
int i = 0
while(buffer[i] != EOF){
    if(buffer[i] == "evaluation"){
        addViolation(report,buffer[i],data)
    }
    i++;
}
```

We have a customized rule where we check for the string in the code “evaluation”, but we don’t mind it in methods. Here is one example of source code with word evaluation as a method and as a printed string.

```
int evaluation(int i) {      // false positive finding
    return i++
}
void write(string s) { ...
    print("evaluation" + s)  // true positive finding
}
```

Due to the simplicity of the rule, we are not able to differentiate between methods and simple strings, and both occurrences will be marked as positive findings. One will be the true positive finding the second one will be the false positive finding. The summary report could look like this:

Finding Report

- Searched methods: 2
  - evaluation
  - write
- Positive occurrences: 2
  - evaluation
    - expl: Violated EvaluationRule
    - src: CodeExample/src.code:1
  - write
    - expl: Violated EvaluationRule
    - src: CodeExample/src.code:5

False positive findings in static analysis occur when the tool reports a potential issue in the code that is not actually a problem. There are several sources of false positives in static analysis, including:

- Complex logic: Static analysis tools may have difficulty understanding complex logic, leading to false positives when they misinterpret code that is actually correct.
- Incomplete information: Static analysis tools only have access to the source code being analyzed and may not have access to all the necessary information, such as environment variables, runtime data, or input/output data. This can lead to false positives when the tool cannot accurately simulate the behavior of the code in a real-world scenario.
- Custom code or frameworks: Static analysis tools are often designed to work with standard coding practices and may not be able to handle custom code or frameworks. This can result in false positives when the tool does not recognize the specific syntax or structure of the code being analyzed.
- Outdated rules: Static analysis tools rely on predefined rules to identify potential issues in the code. If these rules are outdated or not properly configured, they may generate false positives by flagging code that is actually correct.
- Ignored or suppressed warnings: In some cases, developers may ignore or suppress warnings generated by the static analysis tool, leading to false positives when issues are not properly addressed.
- False positives can also be caused by bugs in the static analysis tool itself.

To reduce false positives, it's important to use a combination of automated tools and manual code reviews to identify potential issues.

Chapter 4

Eclipse

Eclipse is Java Interactive Development Environment used in computer programming, it bears the weight of Java development from the programmer, and it is one of the best Java developing programs on the market. It can handle various programming details for the user like syntax checking, error handling, adding imports as needed, targeting builds, commenting out blocks of code with a single click, adding .jar files to the build path, refactoring, and reformatting the code. Eclipse has a steep learning curve, and it can be challenging for many user groups to handle it, it is also distributed for free, and the Eclipse marketplace contains many possible extensions of different programs and functionalities embedded into the Eclipse IDE. The learning curve can be reduced by using Eclipse handbooks, watching YouTube tutorials, or attending beginner's Eclipse Java IDE training courses.

Eclipse uses the Java development toolkit which is a plugin. The Eclipse platform itself is made of basic construction and plugins, making it small and lightweight. Many of the plugins are already preinstalled including JDT. There is a Plug-in development environment, which allows users to develop custom plugins. The entire architecture of the Eclipse program is summarized in Figure 4.1, which describes the connection between each module. Each part of the Eclipse platform including the section on plugins is described below.

4.1 Builds

Program Eclipse is distributed in various builds. There are different characteristics for each category of the build versions.

- Release Build. Released versions, that have been tested and are ready for distribution without any or minimum software bugs. They are for general-purpose use and are in every freshly installed Eclipse IDE.
- Stable Build. Beta versions are called stable builds. They provide the user with upcoming features and are still clean of bugs. There is a slightly higher possibility of running into a problem or finding a bug in the program. The user has to choose to receive these kinds of updates.
- Integration Build. One step below stable builds are integration builds, where the new components have been well tested, but users still can run into bugs. If everything is alright, these builds are made into Stable versions.

- Nightly Build. Builds of this type are risky and are the newest experimental available updates that the Eclipse team can provide. The is a high probability that the user will run into some misconfigurations or failures in the program and can encounter bugs while using it.

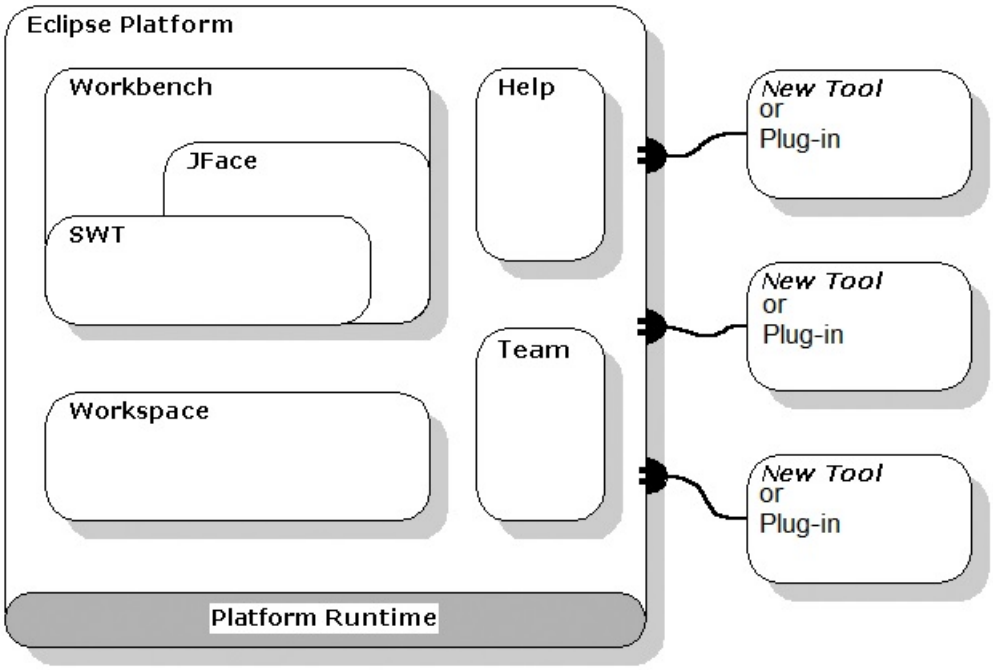


Figure 4.1: Eclipse Architecture [6]

4.2 The Eclipse Platform Kernel

The kernel is written in Java, it warns the user if the correct Java installation is not present for running the rest of the program. It is also responsible for loading plugins. [12]

4.3 Workbench Component

The Eclipse workbench serves as a user interface for the platform, displaying menus, toolbars, and plugins in the style of the operating system. It provides toolkits such as SWT and JFace, which allow Eclipse to interact with the underlying operating system and provide user interface elements like fonts, dialogues, and wizards. SWT enables Eclipse to use native widgets wherever possible while emulating them for unsupported platforms.

JFace is a toolkit built on top of SWT, offering features like action and viewer mechanisms. These mechanisms standardize behavior for widgets such as lists, trees, and tables, providing consistent and predictable interactions for users. Viewers keep widgets synchronized and handle common behavior like filtering and sorting. Providers can be used to map input, labels, and icons to displayed elements, allowing for greater customization and flexibility. [6]

4.4 Workspace Component

The Workspace is a central component of Eclipse IDE that manages project resources, including code and files, and stores their history of changes. It provides an interface for accessing and manipulating project assets and working with plugins to customize its functionality. The Workspace allows users to annotate resources with customizable markers and track changes to project resources through an event notification system. It also includes an incremental project builder framework for the analysis and transformations of many files and enables plugins to be used consistently across sessions. Overall, the Workspace plays a crucial role in enabling developers to work efficiently and collaboratively within the Eclipse IDE. [6]

4.5 Team Component

Source code control and repositories play a big role while developing in corporate companies. Eclipse is provided with a source code control platform called Concurrent Versions System allowing developers to manage their repositories. [12]

4.6 Help Component

It is an extensible help component integrated into Eclipse. Most plugins provide their help in XML formats to direct the system on how to navigate through their documentation. [12]

4.7 Plugin Component

Plug-in, add-on, add-in, or simply extension is a software program used for enhancing the capabilities and functionality of a specified program. There are many types of plugins, that could be divided into paid plugins and free plugins. Paid plugins are program extensions that the user has to pay for, to use them. Free plugins are distributed without any additional fee providing the user with their functionality. Plugins can be obtained as a preset package ready to be automatically installed, or as a part of code downloaded from the repository, needed manual installation. They can alter the visual feedback given by the program to the user by changing its user interface, adding new functionality in the form of device drivers, and many more possibilities. [12]

4.8 Eclipse Plugins

New tools or Plug-ins are part of the Eclipse framework. The integrated development environment (IDE) itself is full of plugins, which allow the user to edit an Ant file, compile a Java file, drag and drop GUI elements, change perspectives, use different editors, have more views, wizards, and more. There are more than 1000 tools and plugins available for the Eclipse IDE. There are many plugins for static analysis, and some of them will be described.[12]

Languages are distributed in Eclipse via plugin fragments which makes it possible for numerous languages to be used in the interactive development environment such as Japanese, Korean, German, English, French, Italian, Portuguese, Spanish, traditional or simplified Chinese, and even Czech, which is distributed in different language package. [11]

In the following sections, we give a brief overview of static analysis plugins.

4.9 FindBugs

FindBugs is a static analysis tool based on the concept of bug patterns. It uses the scanning technique of the code, where it scans over Java bytecode and tries to map predefined rules to find a match and therefore the bugs. See what predefined rules in the first section of static analysis are. Java bytecode is compiled class files, therefore the tool does not need the source code of the application in order to perform the analysis. The bugs could be misunderstood API methods, invariance in the code caused during the maintenance, or simple typos in the code. It then reports bugs in several categories, listed by their ranks such as scariest, scary, troubling, or of concern, which could be set to be shown as an error, warning, or info. There is also a minimum rank to report ranging from 1 to 20, where 1 is the most severe with a minimum confidence report with three levels, low, medium, and high. The categories for the bugs are bad practice, malicious code vulnerability, correctness, performance, security, dodgy style, experimental, multithreaded correctness, and internationalization. This tool functions as a stand-alone or plugin in various programs like Eclipse, IntelliJ IDEA, Gradle, Hudson, Maven, Bamboo, Jenkins, and Netbeans. Its big advantages over other tools like PMD, since it works with compiled class files, are: equals() method fails on subtypes, clone method may return null, reference comparison of Boolean values, impossible cast, 32bit int shifted by an amount, not in the range of 0-31, a collection which contains itself, an infinite loop, etc. FindBugs is currently an abandoned project. [3, 24, 33, 8]

4.10 Spotbugs

Spotbugs is a static code analysis tool that is a successor of the tool FindBugs, and it inherits all advantages and functionality of its predecessor. It can be used as a stand-alone or in Eclipse, Maven, Gradle, or Ant. It can detect over 400 bugs such as bad practice, finalizer nulls fields, very confusing methods names, Method does not override the method in the superclass due to the wrong package for the parameter, and Uninitialized read of field method called from the constructor of the superclass, and many more. The tool has its extensions which are the most popular A FindBugs auxiliary detector plugin (fb-contrib) and Find Security Bugs plugin (find-sec-bugs). [33, 29, 5]

4.11 PMD

Is an extensible cross-language static code analyzer. It focuses mainly on the Java and Apex programming languages, but it also supports 12 other programming languages. It scans the source code for typical programming mistakes, like unused variables, empty catch blocks, unnecessary object creation, and more. It does this with predefined rules, which can be customized to the will of the user. PMD has many possibilities to be integrated into tools such as Maven, Gradle, Ant, BlueJ, Eclipse, IntelliJ IDEA, JCreator, and more. The main plugin advantages are copy-paste detector (CPD), which can find duplicate code, various types of reports, and different rule configurations. It can find bugs like a violation of naming conventions, lack of curly braces, misplaced null check, long parameter list, unnecessary constructor, missing break in switch, etc. PMD tool also reports on the cyclomatic complexity of the code, which on the other hand FindBugs tool does not do. [20, 25, 21, 8]

4.12 JDepend4Eclipse

JDepend is a Dependency analysis tool used for scanning Java packages in a project. It can be a great help when refactoring large and old parts of source code repositories with cyclic package dependencies between packages. It creates a set of metrics for each Java package from scanned Java classes and source directories. Dependency analysis recursively searches for dependencies between artifacts, which then adds to the list of dependencies. If the dependencies are dependent on another artifact, that means they are indirectly dependent on the artifact, also these will be included. This algorithm can create a dependency graph, starting with the chosen artifact. JDepend is a plugin used in various tools such as Jenkins, Maven, Ant, or Eclipse. It can provide the user with information about packages with dependency cycles, and direct and indirect dependencies. Users may choose to use this tool for scanning dependencies also because it can automatically measure the quality of a design in terms of extensibility, reusability, and maintainability. [14] [7] [8]

4.13 CodeMR

This is a multi-language software tool made especially for programming languages Java and C++ with software quality checking and static code analysis. It checks for coupling, cohesion, complexity, and size of the program. You can visualize different kinds of Java classes such as in package structure, treemap, sunburst graph, or in dependency and graph view. With the graphical visualizations, you can check relations between each Java class. CodeMR offers the user many different filtering options and supports custom queries and can aggregate results into simple graphs for the user to review and save the resulted metrics in different formats such as HTML web page, png, mdl, cmr, and more. It can be integrated into Eclipse or IntelliJ IDEA. Its biggest advantage over other software quality tools is that it can provide the user with numerous types of metrics such as method metrics, class metrics, project metrics, and package metrics like the number of packages, package instability, efferent coupling, cyclomatic complexity, or coupling between object classes. Its disadvantage is that on its free version, it has limited extraction to 50 source files and 60 classes. [28] [8]

4.14 Summary

The plugins we have mentioned above are powerful tools that can help the developers thoroughly test code and get clear results, however, our evaluation aimed to determine if it was better to develop extensions to one of these plugins or to proceed with the implementation of extensions to the MFB Analyzer. After a thorough evaluation, we found that the MFB Analyzer is the ideal choice for testing the targeted architecture. This is because it was built specifically for this architecture and is designed to test the APIs of both the main application and its extensions. In contrast, the other static analysis plugins we evaluated have completely different core functionality, and developing extensions for them would not be suitable for our specific needs. Therefore, we decided to proceed with the design and implementation of extensions to the MFB Analyzer. This decision was based on the fact that it is better suited to our needs, and we are confident that it will provide us with the best possible results. Additionally, we plan to independently implement an Eclipse marking plugin to highlight the results obtained from the MFB Analyzer.

Chapter 5

Targeted Code

The targeted system is a product of BOC-GROUP that works with the Business Process Model and Notation (BPMN). 5.1 BPMN is a graphical notation used for designing and documenting business processes. It provides a standardized way of representing business processes graphically, enabling process designers to create diagrams that are easy to read and understand.

BPMN provides a common language for business analysts, IT professionals, and other stakeholders to communicate and collaborate on the development and improvement of business processes. It enables organizations to model and automate their workflows, monitor process performance, and make data-driven decisions to optimize their operations.

Some of the key features of BPMN include the ability to define process flows, manage process data, specify rules and constraints, handle exceptions and errors, and integrate with other enterprise systems. It is often used in conjunction with Business Process Management (BPM) software, which provides a platform for designing, executing, and monitoring business processes using BPMN.

5.1 BOC-GROUP Applications

The product of BOC-GROUP is three main systems called ADONIS 5.2 specialized in business process management, ADOIT for enterprise architecture, and ADOGRC for governance, risk, and compliance. We will focus mostly on ADONIS and its extensions.

ADONIS is a powerful Business Process Management (BPM) tool offered by BOC-GROUP. It allows organizations to model, analyze, optimize, and automate their business processes with ease. ADONIS follows the Business Process Model and Notation (BPMN) standard, which is widely accepted as the industry standard for process modeling.

With ADONIS, users can model complex workflows using a user-friendly graphical interface that conforms to international standards such as ISO 9000. It offers a broad range of functionality in areas such as process optimization, quality management, process simulation, and more.

ADONIS supports the implementation of process improvement initiatives like Business Process Re-engineering (BPR) and Continuous Improvement (CI) by providing tools for process analysis, redesign, and implementation. Additionally, the system includes features for managing quality standards and ensuring compliance with industry standards.

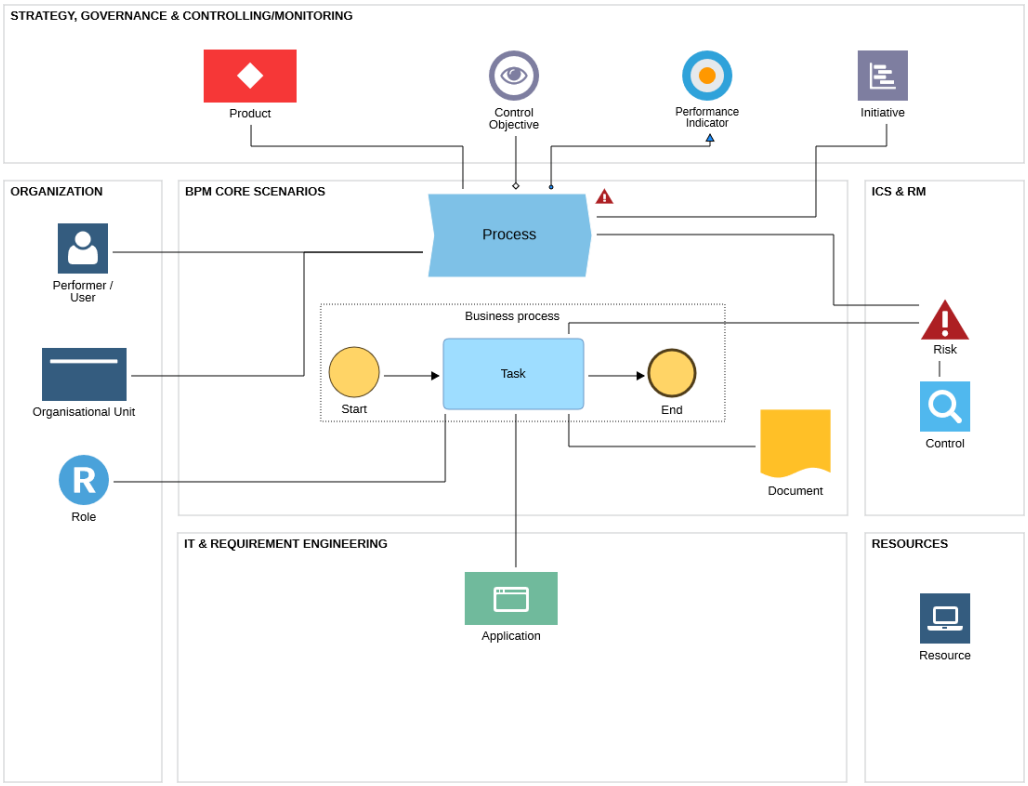


Figure 5.1: Business Process Model Scenarios [1]

Users can simulate and test their process models using ADONIS, providing insights into process performance and enabling data-driven decision-making. The system also provides extensive reporting and documentation capabilities for generating reports and documenting processes. [1, 32]

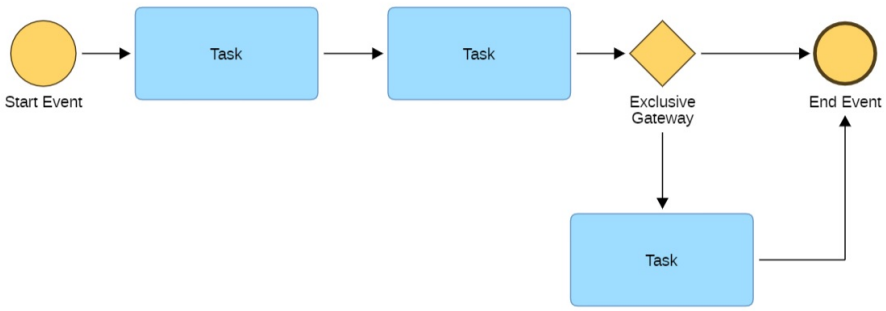


Figure 5.2: Flow Example from ADONIS

5.2 BOC-GROUP Application Extensions

ADONIS is a versatile tool that offers three main scenarios: Design and Document, Control and Release, and Read and Explore. In this discussion, we will focus on the Design and Document scenario, which allows users to design business processes and model their workflows.

In addition to the base architecture and functionality, ADONIS has custom plugins called method and functional building blocks (MFBs), which extend the functionality of the ADONIS core. These MFBs are tested with the MFB Analyzer and connected to the main application via APIs. The MFBs offer a range of features, including Excel export, PDF export, process simulation, custom web application connectivity, and SAP integration.

The MFBs contain public, private, and deprecated Java and javascript methods and classes. To ensure proper implementation of APIs, the custom solution of the MFB analyzer is used instead of relying on results concerning Java code.

Additionally, ADONIS features the Control and Release module, which manages roles and personas, allowing for streamlined management of user access and permissions.

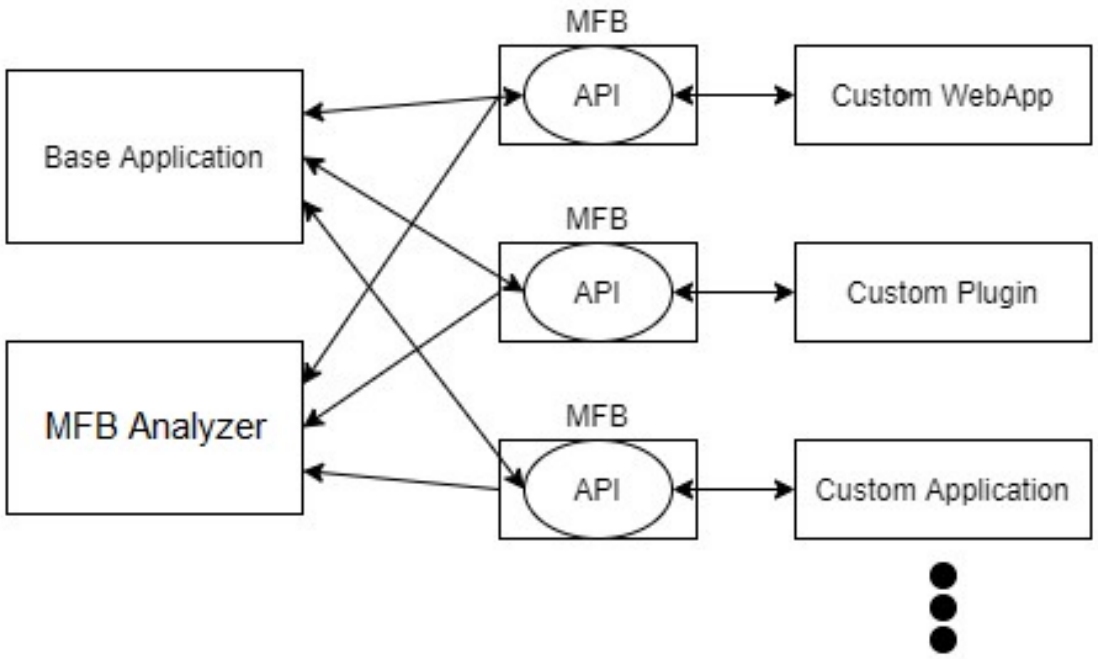


Figure 5.3: App Architecture

5.3 Testing

During the development phase of any software application, it is crucial to ensure that all components work seamlessly together. This is especially important in the case of ADONIS, where the main application communicates with its extensions through APIs. Since the development of the main application and its extensions occur independently, there is a risk of breaking the communication link between them.

To address this challenge, ADONIS uses a custom-developed analyzer called the Method and Functional Building Blocks (MFB) Analyzer. The MFB Analyzer is designed to test the APIs used by the extensions and ensure that they continue to work correctly even after updates or alterations.

The MFB Analyzer tests the public, private, and deprecated Java and javascript methods and classes within the MFBs. It checks for potential vulnerabilities and bugs in the code that could potentially break the API communication link between the main application and the extensions.

By using the MFB Analyzer, ADONIS ensures that the main application and its extensions are compatible with each other and can communicate seamlessly. This enables ADONIS to provide a reliable and robust platform for its users to model, analyze, optimize, and automate their business processes with ease and confidence.

Chapter 6

MFB Analyzer

6.1 MFB Parser

BOC-group has developed a powerful static analysis tool for Java and JavaScript languages known as the MFB Analyzer. This tool is composed of two static analyzers, namely the MFB Parser 6.1 and Security Checker 6.1, which work together to identify potential issues in the codebase.

The MFB Analyzer relies on a third-party library called JavaParser to extract the names of methods and their parameters from the Java code. This library helps the analyzer to build a comprehensive understanding of the codebase, enabling it to identify potential vulnerabilities, performance issues, and other problems.

In addition to the Java analysis, the MFB Analyzer includes two additional files for checking against JavaScript classes and methods. This feature extends the functionality of the tool to include JavaScript and provides a comprehensive analysis of both Java and JavaScript codebases.

Overall, the MFB Analyzer is a powerful static analysis tool that can help developers identify potential issues in their codebase. With its advanced features, it is a valuable tool for ensuring code quality, reducing the risk of vulnerabilities, and improving the overall performance of applications.

6.2 Security Checker

The Security Checker, an essential component of the MFB Analyzer developed by BOC-group, is designed to thoroughly examine the source code for potential security vulnerabilities. It analyzes the code for explicit strings and categorizes them based on their severity, placing them into categories such as Audit, Bugs, Critical, and Highlights. Related to the chapter 3, security checker categories contain different sets of rules by which they obey and test the code such as forbidden strings, XML handling, File handling, and if correct encryption functions are used.

The Audit category executes the npm audit to identify any third-party libraries that may have security issues. The Critical category focuses on high-risk areas such as XML handling, REST interfaces, and Java code, where vulnerabilities can pose a significant threat to the application. The Highlights category flags issues that could potentially become a security issues in the future, such as using weak encryption algorithms or passwords.

The results of the Security Checker are combined with those of the MFB Parser and summarized in a JSON format. This JSON file is then converted into a user-friendly webpage that displays the results in a convenient and easy-to-understand manner. Although the Security Checker is a useful tool, it does not currently exist as an Eclipse plugin.

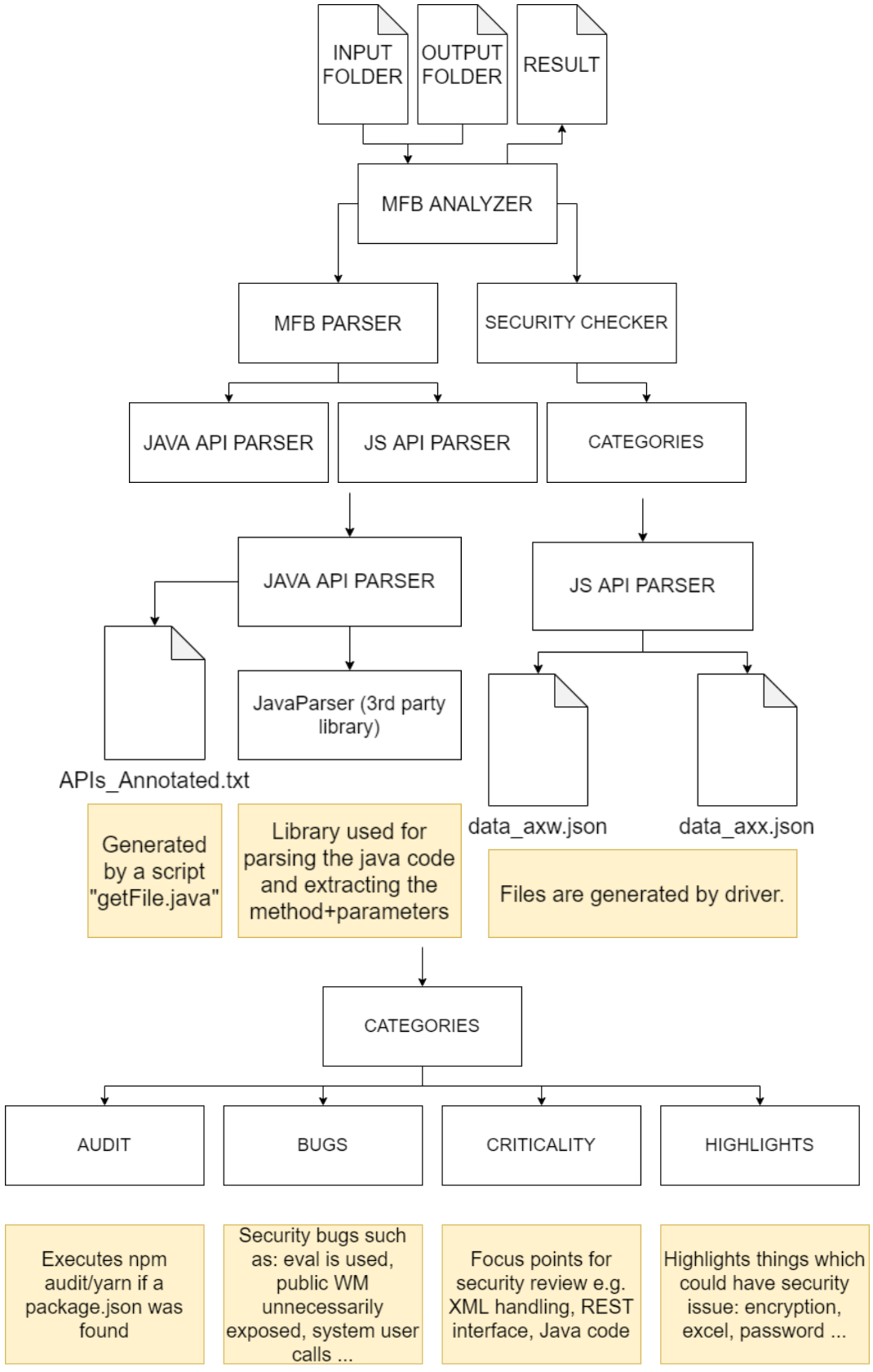


Figure 6.1: MFB Analyzer architecture

Chapter 7

MFB Analyzer Extensions

7.1 Introduction

In this chapter, we will dive into the Log4j library and its vulnerabilities that are specifically targeted by the MFB Analyzer extensions. We will build upon the previous discussions about the Targeted Code and MFB Analyzer to provide an in-depth understanding of Log4j. Additionally, we will discuss the General Data Protection Regulation (GDPR) and its significance for organizations, as well as the potential consequences of data leakage for both individuals and organizations and the problems we faced during its implementation. We will describe how we designed and implemented the GDPR Analyzer extension, which uses regular expressions to detect sensitive data leaks in Java code. The GDPR Analyzer serves as a valuable tool for developers to ensure that they are complying with GDPR regulations and protecting user privacy. The Log4j and GDPR Analyzers are just two examples of the numerous extensions that can be integrated into the MFB Analyzer, providing a wide range of functionalities for developers to improve their code quality and security, more examples will be discussed in the last chapter Possible Extensions.

7.2 Log4j

Log4j is an open-source logging utility library for Java programming. This library is designed to log messages of various severity levels to different outputs such as console output, files, databases, etc. It also provides a flexible mechanism for configuring logging behavior. It has several advantages over the System.out.println debugging. It allows developers to easily change the logging at runtime without requiring code changes or restart. It also allows messages to be logged to multiple output targets, making it easier to manage and analyze logs. And it provides a large set of configurations that can be used to customize logging behavior.

Log4j allows developers to log messages of varying levels such as DEBUG, INFO, WARN, ERROR, and FATAL. They help developers more effectively manage and troubleshoot issues in the application. For example, DEBUG severity can help trace information in the application, warning messages can be used to indicate potential issues which do not require immediate attention, while error and fatal messages can indicate critical problems in the application which need immediate attention and fixing.

The biggest benefit of using this library is in faster troubleshooting and resolution times and improved application performance.

7.3 Log4j Exploit

Referring to the officially known CVE-2021-44228 Log4j exploit. It is a critical security vulnerability that affects the Log4j library. The vulnerability was first discovered in December 2021 and it affects all versions of Log4j up to and including version 2.15.0 and it has been given the highest severity rating of 10.0 on the CVSS (Common Vulnerability Scoring System) scale. The vulnerability got its attention due to the widespread usage of Log4j.

This exploit is particularly dangerous because it allows attackers to remotely execute arbitrary code on a targeted system. This can be done without authentication or user interaction. Once the system has been exploited attackers can gain complete control over the targeted system, access sensitive information and carry out malicious activities.

The exploit takes advantage of a flaw in the library's handling of certain types of log messages. The attacker can exploit the Log4j vulnerability by sending a specially crafted log message to a vulnerable application that uses the affected version. The log message can include a specific code sequence that triggers the vulnerability, when the application tries to log the message, the payload can be commands that can be executed on the affected system. The payload message could be for example:

```
String payload = "${(\"\".getClass().forName(\"java.lang.Runtime\")
        .getMethod(\"getRuntime\").invoke(null)).exec(\"calc\")}");
```

When this code would be executed on a vulnerable Log4j system, the code would trigger the payload during the message logging, which would result in launching the Windows calculator application. [9, 36]

7.4 Log4j Checker Extension

Log4j is a popular library and is usually included in Java projects in their dependencies and in Java classes using the functions from the library. This makes them more detectable by the static code analyzers and dependency checkers. We have decided in the implementation to use the approach of searching for Log4j libraries directly in the source code of the Java classes since our Analyzer (MFB Analyzer) works only with the source code. That is the target of our analyzer extension and the static analysis.
Here are the steps of the checker:

- The Analyzer creates a list of paths to the tested files
- The Log4j Extension reads each file
- Checks the text with regular expressions for Log4j libraries
- If found, returns specific information about the finding (row, col, path, etc.)
- Creates a result file

Here are some examples of included Log4j library in Java projects:

- `import org.apache.log4j.BasicConfigurator;`
- `import org.apache.log4j.Logger;`

- `package org.apache.log4j`

- ```
  <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.11<version>
  </dependency>
  ```

In the Log4j extension of the MFB Analyzer, we have implemented specific rules for static analysis to identify the dependencies of the Log4j library in the projects.

Here are some simple rules for the static analysis:

- FindLog4jImport

  ```
  "import\\s+org\\.apache\\.logging\\.log4j\\..+"
  ```

- FindLog4jNew

  ```
  "new\\s+org\\.apache\\.logging\\.log4j\\..+"
  ```

- Findlog4jDependency

  ```
  "<dependency>.*<groupId>org\\.apache\\.logging\\.log4j<\\/groupId>
      .*<artifactId>log4j-api<\\/artifactId>.*<\\/dependency>"
  ```

- FindLog4jImplementation

  ```
  "implementation\\s+group:\\s+['\"]org\\.apache\\
      .logging\\.log4j['\"],\\s+name:\\s+['\"].*['\"]"
  ```
  ```

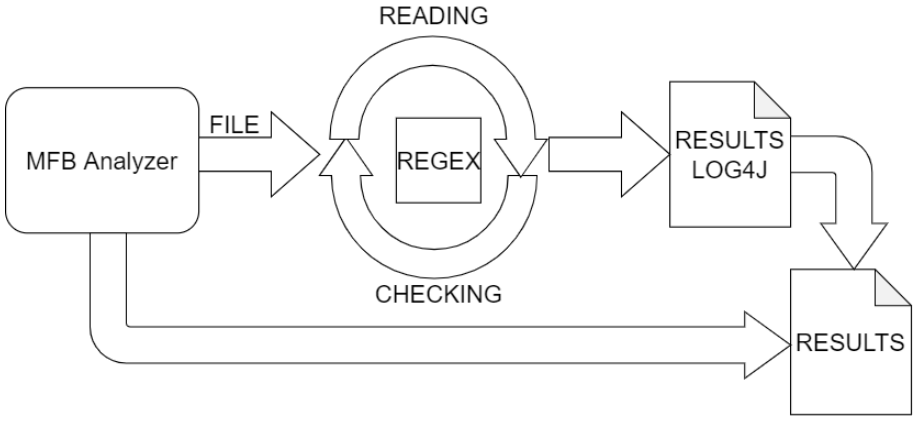


Figure 7.1: Log4J checker

The results and the Log4j results are very important for the Eclipse Annotation Plug-in. They will be summarized and directly shown in the Eclipse IDE.

There are many programs on the market that are able to detect Log4j vulnerability such as Snyk, Nessus, or Blackduck. But any of the programs would be easy to integrate into the Analyzer and its results. The main benefits are summary in one page, customization, and handling for the developers of the BOC-Group, and control over the tool and the vulnerability detection process.

7.5 GDPR

The General Data Protection Regulation (GDPR) is a comprehensive data protection law that regulates how the personal data of individuals are collected, processed, and stored by businesses, organizations, and government bodies in the European Union. The regulation was introduced in Europe in May 2018. [30]

The regulation applies to all companies that process the personal data of individuals within the EU. That means that any company which collects, processes, or stores any personal information of individuals in the EU must comply with the regulation. Personal data is any information that can be used to identify an individual such as their name, address, email, phone number, IP address, and other sensitive information.

The GDPR enhances individuals' privacy rights and strengthens the obligations of businesses to protect personal data. It also helps unify data protection laws across the EU, making it easier to operate across different member states.

Businesses must receive individuals' informed consent before processing their data. Individuals must be given clear information about how their data will be used and they must give their consent. Businesses must also provide them with the right to withdraw their consent at any time.

Individuals have the right to request access to the data that is being processed by a business or organization, that includes categories of data being processed, the purpose of processing, and any recipients of the data. Right, to request to correct or update any inaccurate or incomplete personal data. Right to be forgotten, that is to request their data to be deleted. Right to object to the processing of their data including for direct marketing purposes. Right to restrict processing, to data portability, and right to not be subject to automated decision-making including profiling that has legal or significant effects on them.

Companies are required to implement appropriate technical and organizational measures to protect personal data. This includes encryption, access controls, and backups. They must ensure that any third-party processors they use are compliant with the GDPR. Companies are also required to report any data breaches to individuals and protection authorities within 72 hours. If the business fails to comply with the GDPR, it can result in significant fines with penalties of up to 4% of a company's annual global revenue or €20 million. As such it is important for businesses to take GDPR seriously and implement effective data protection measures to comply with the regulation. [30]

7.6 GDPR Leak

As mentioned previously non-compliance with the GDPR legislation or data breaches are very dangerous for companies and individuals, here is why. Sensitive personal information such as names, addresses, social security numbers, and credit card numbers can be used by cybercriminals to steal an individual's identity and commit a crime. If financial information is leaked, it can result in unauthorized access to bank accounts and credit cards leading to financial loss of individuals and organisations. Organizations that experience data breaches may lose their reputation if customers and the public lose trust in their ability to secure their personal data. Organisations may face legal and regulatory consequences including fines and legal actions by affected individuals. When the leak is related to national security it can pose a risk to the safety and security of governments and societies.

7.7 GDPR Checker Extension

The BOC-GROUP company values the privacy of individuals and takes it very seriously. However, many organizations have faced issues with the GDPR when processing personal data after obtaining consent from users, specifically with regard to collecting IP addresses and other personal information through web server logs. These organizations have failed to collect this data correctly, leading to the storage of sensitive information such as usernames and email addresses without adequate anonymization or pseudonymization. This creates a risk to the privacy of individuals, highlighting the importance of correctly managing web server logs and ensuring compliance with the GDPR. [13]

Due to this issue and the importance thereof, we have decided to implement the GDPR checker as the Analyzer extension (MFB Analyzer). The implementation uses the approach of searching for sensitive data which can be contained in variables directly in the source code of the Java classes since our Analyzer works only with the source code. That is the target of our analyzer extension and the static analysis.

Here are the steps of the GDPR checker:

- The Analyzer creates a list of paths to the tested files
- The GDPR Extension reads each file
- It cleans the processed text of any misleading data in the targeted logging function
- Checks the data with regular expressions for leaks of any sensitive information
- If found, returns specific information about the finding (row, col, path, etc.)
- Creates a result file

The Extension targets code functions that create any type of log in the application. These logging functions concatenate strings and variables with a string representation and stream them into a file or to a standard output. All the variables containing any sensitive data must be encrypted, this can be checked by cleaning the insides of the logging function of any unnecessary text such as constant Strings or comments. In the end, we check if any sensitive data is encrypted. If these conditions are not met, it is marked as a positive finding and returned information about the issue such as line number, file name and path, and other important descriptions.

Here are some simple rules for static analysis using regular expressions checking code for any sensitive information:

- **FindAnyUserOrUsername** which checks if any user or username has been detected

  ```
  "(?i)^(s|usr|username|myuser)[A-Za-z0-9]*$"
  ```

- **FindAnyPassword** which checks if any password has been found

  ```
  "(?i)^(p|pass|password)[A-Za-z0-9]*$"
  ```

- **FindAnyEmail** which checks if any email has been found

  ```
  "(?i)^(e|email|emailaddress|myemail)[A-Za-z0-9]*$"
  ```

- **FindCreditCard** checks for possibility of credit cards

  ```
  "(?i)^(c|creditcard|mycredit)[A-Za-z0-9]*$"
  ```

- **FindAnySsnOrSocialSecurityNumber** checks for ssn

  ```
  "(?i)^(ssn|socialsecuritynumber|myssn)[A-Za-z0-9]*$"
  ```

The results and the GDPR results are very important for the Eclipse Annotation Plug-in. They will be summarized and directly shown in the Eclipse IDE.

There are many programs on the market that are able to detect sensitive data leaks such as Veracode, IBM Security AppScan, or Fortify. But any of the programs would be easy to integrate into the Analyzer and its results. The main benefits are summary in one page, customization, and handling for the developers of the BOC-Group, and control over the tool and the vulnerability detection process.

7.8 Summary And Roadblocks

During the research and implementation phase of the Log4j and GDPR Analyzer extensions, we encountered several challenges that required extensive time and effort to overcome. The most significant obstacle was understanding the structure of the MFB Analyzer and how

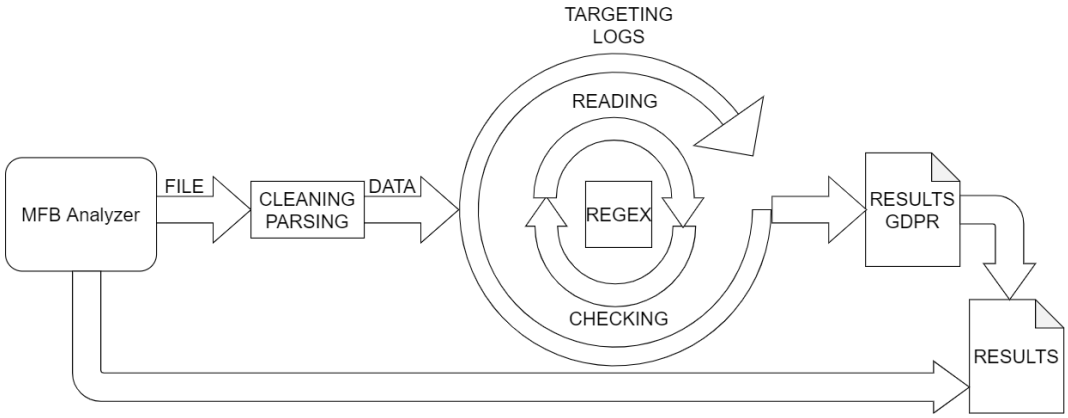


Figure 7.2: GDPR checker

we could incorporate the extensions into it. This required us to conduct extensive research into the analyzer's functionality and design, which was not well-documented and known to only a few individuals. However, through trial and error, we were able to gain a clear understanding of the analyzer and identify the best places to integrate our extensions.

The implementation of the Log4j extension was relatively straightforward, as we were able to leverage the library's existing vulnerabilities and apply our static analysis rules to detect them. However, the GDPR extension posed more significant challenges. We needed to parse through many conditions to check for our static analysis rules, which was time-consuming and required significant effort. Moreover, we found that the rules were not perfect, as variables hidden under different labels were challenging for the extension to identify.

Despite these challenges, we believe that the BOC-GROUP company will benefit significantly from using these extensions. They provide extended security against Log4j exploits and GDPR data leaks. Additionally, the custom solution we have developed will give the developers agility in analyzing the code and help them to pinpoint problematic code areas. The extensions provide valuable information about the line, description, and file names of the problematic code, making it easier for developers to fix issues quickly and efficiently.

Chapter 8

Eclipse Marking Extension

8.1 Introduction

In this chapter, we will explore the design and implementation of a marking system for Eclipse IDE that can be used to display the results of a code analyzer tool. We will discuss the challenges we faced during the implementation and how we overcame them. Additionally, we will provide examples of how to create and manage markers using the Eclipse Marker API and discuss best practices for using markers to improve code quality and developer productivity.

8.2 Markers

Markers are an important part of Eclipse IDE, they annotate and highlight lines of code or resources with specific information such as warnings, errors, and bookmarks, all with descriptions. They provide feedback to the user about the status of the IDE. They are associated with specific resources, files, or projects and can be used to indicate issues with those resources. For example, an error marker can be used to indicate a potential syntax error in a Java file. They can also indicate progress or status. They can be customized and tailored to the developer's needs using various attributes, such as severity, message, location, and icon. The appearance and behavior of markers can also be customized using preferences or plugins.

Markers can be created and managed programmatically using the Eclipse APIs, which allow plugins and extensions to interact with the markers and provide custom marker types, which we will use in the extension.

Here is a short example of how to create a new marker in the Eclipse IDE:

```
// Get the current workspace
IWorkspace workspace = ResourcesPlugin.getWorkspace();

// Get the Java file in the workspace
IResource javaFile = workspace.getRoot()
        .getFileForLocation(new Path("path/to/JavaFile.java"));

// Create a new warning marker
IMarker marker = javaFile.createMarker(IMarker.PROBLEM);
marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_WARNING);
```

```
marker.setAttribute(IMarker.MESSAGE, "Hello World");
marker.setAttribute(IMarker.LINE_NUMBER, 10);
```

In this example, a new marker is created using the IMarker interface. It is created with these attributes. It is a PROBLEM with a severity of the WARNING and after showing it displays „Hello World“ in the description. It is located in a file called JavaFile.java at line number 10. When a new Eclipse window with this file is opened, the program will automatically register the marker and displays the annotation to the user.

8.3 Markers Implementation

The MFB Analyzer is a tool that analyzes code and produces results in the form of JSON files. These files contain detailed information about any problems detected during the analysis, as well as the location of the problem. The Eclipse extension for the MFB Analyzer takes each result file produced by the Analyzer and extracts the relevant information. This includes the location of the problem, the severity of the problem, and any other information that is necessary for developers to understand and resolve the issue.

Figure 8.1: Marker line

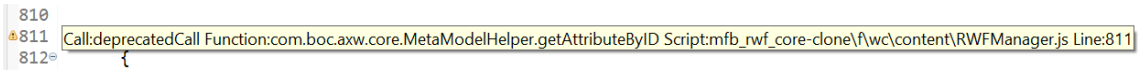


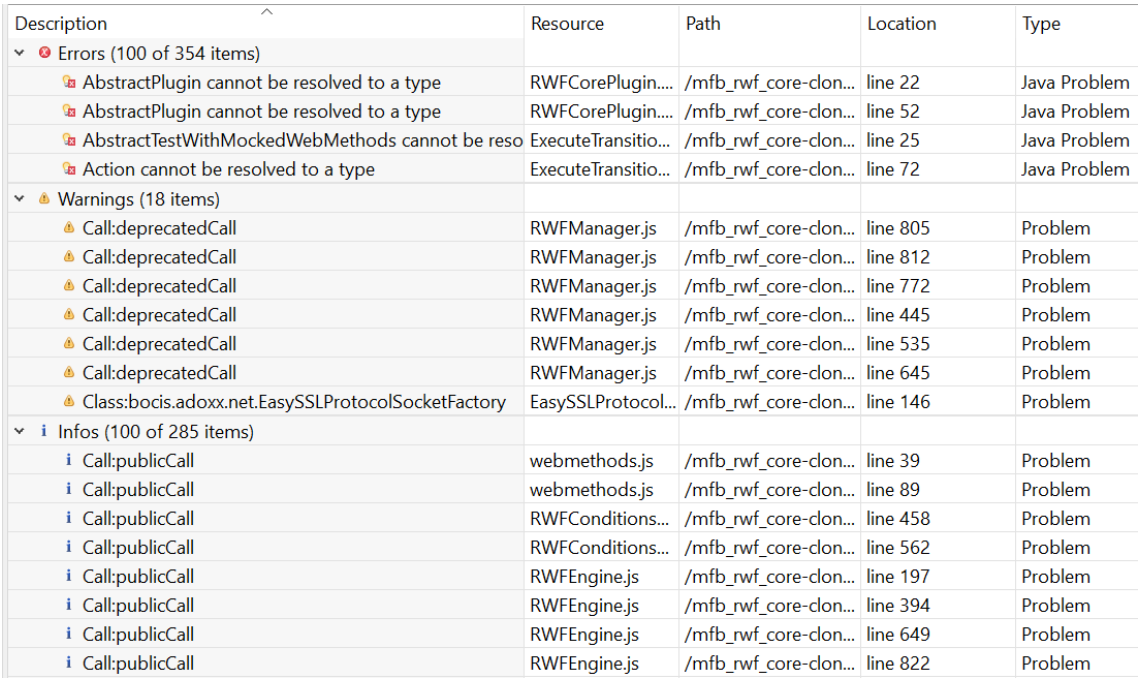
Figure 8.2: Marker line with description

The extension then creates markers of different severity levels, depending on the nature of the problem. Once the markers have been created, the extension loads them into the Eclipse resources and highlights the lines where the problems were detected. 8.4 This makes it easy for developers to locate and fix the issues, without having to search through their code line by line. The markers are also displayed in the Eclipse Problems view, which provides developers with a summary of all the issues detected by the Analyzer. 8.3 This integration allows developers to quickly and easily access the results of the Analyzer, without having to leave their IDE, improving their efficiency and focus on resolving issues. 8.1 8.2

8.4 Summary And Roadblocks

During the implementation, we encountered significant challenges while working with Eclipse, which proved to be a complex and poorly documented platform. We devoted a significant amount of time to researching how Eclipse works internally and how to manage the Eclipse workbench, including the inclusion of plugins, which are described in chapters 4 and 5.

To expedite the development process, we drew inspiration from several open-source projects on GitHub, such as Eclipse-HighlightOnSelection by Huntingriver. Despite this, the majority of our time was spent researching and troubleshooting how Eclipse works, rather than developing the actual marking system.



Description	Resource	Path	Location	Type
Errors (100 of 354 items)				
AbstractPlugin cannot be resolved to a type	RWFCorePlugin....	/mfb_rwf_core-clon...	line 22	Java Problem
AbstractPlugin cannot be resolved to a type	RWFCorePlugin....	/mfb_rwf_core-clon...	line 52	Java Problem
AbstractTestWithMockedWebMethods cannot be reso	ExecuteTransitio...	/mfb_rwf_core-clon...	line 25	Java Problem
Action cannot be resolved to a type	ExecuteTransitio...	/mfb_rwf_core-clon...	line 72	Java Problem
Warnings (18 items)				
Call:deprecatedCall	RWFManager.js	/mfb_rwf_core-clon...	line 805	Problem
Call:deprecatedCall	RWFManager.js	/mfb_rwf_core-clon...	line 812	Problem
Call:deprecatedCall	RWFManager.js	/mfb_rwf_core-clon...	line 772	Problem
Call:deprecatedCall	RWFManager.js	/mfb_rwf_core-clon...	line 445	Problem
Call:deprecatedCall	RWFManager.js	/mfb_rwf_core-clon...	line 535	Problem
Call:deprecatedCall	RWFManager.js	/mfb_rwf_core-clon...	line 645	Problem
Class:bocis.adoxx.net.EasySSLProtocolSocketFactory	EasySSLProtocol...	/mfb_rwf_core-clon...	line 146	Problem
Infos (100 of 285 items)				
Call:publicCall	webmethods.js	/mfb_rwf_core-clon...	line 39	Problem
Call:publicCall	webmethods.js	/mfb_rwf_core-clon...	line 89	Problem
Call:publicCall	RWFConditions...	/mfb_rwf_core-clon...	line 458	Problem
Call:publicCall	RWFConditions...	/mfb_rwf_core-clon...	line 562	Problem
Call:publicCall	RWFEngine.js	/mfb_rwf_core-clon...	line 197	Problem
Call:publicCall	RWFEngine.js	/mfb_rwf_core-clon...	line 394	Problem
Call:publicCall	RWFEngine.js	/mfb_rwf_core-clon...	line 649	Problem
Call:publicCall	RWFEngine.js	/mfb_rwf_core-clon...	line 822	Problem

Figure 8.3: Problems view

One of the challenges we encountered during the implementation was related to the Eclipse IDE's struggle with overlapping libraries, which slowed down the process. However, once the fundamental issues were resolved, we were able to design and implement the marking system logic into the Eclipse IDE using the results from the MFB Analyzer.

In conclusion, the Eclipse marking plugin has proven to be a valuable tool, particularly during the testing of the MFB Analyzer extensions. It has significantly reduced the amount of time spent searching through code and result files, thus helping developers save time and minimize human error. We believe that developers will greatly benefit from using this tool in their work, as it streamlines the process and provides a visual representation of the Analyzer's results directly in the code.

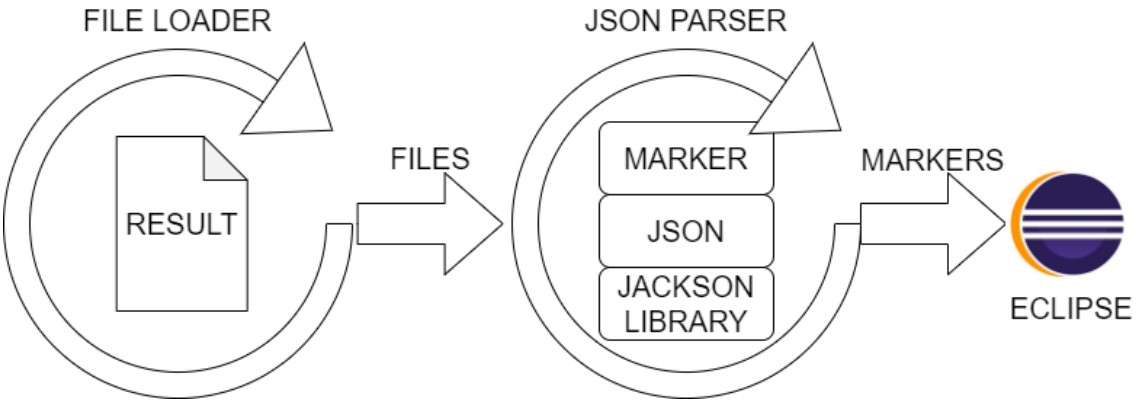


Figure 8.4: Annotation diagram

Chapter 9

Possible Extentions

9.1 Introduction

In this chapter, we will explore ways to enhance the functionality of the MFB Analyzer and its plugin by incorporating static analysis techniques to provide a more robust security solution. We will examine various types of cyberattacks and their potential impact on both individuals and organizations, highlighting the importance of proactive security measures. Through the use of static analysis, we can identify vulnerabilities in software code before they can be exploited by attackers, therefore minimizing the risk of data breaches and other cyber threats. Each section of this chapter will delve into different aspects of static analysis and offer practical strategies for improving system security.

9.2 Harmful Regular Expression Detection

The regular expression (regex) could be a weak point in the application especially when they are receiving unchecked inputs. In this instance, it can lead to an attack called regular expression denial of service (ReDoS). It is a type of Denial of Service attack which exploits this type of vulnerability. The attacker can make the program execute some code through the regular expression and make the service very slow.

The problem lies in the regex itself since the regex algorithm builds a finite nondeterministic state machine. When the engine starts, it searches paths from the first state to the end state and explores all possible paths until a match is found, or all paths have been explored, then the match fails. The major problem in regex is called backtracking. It is a traversal method, when an input (token) fails to match, the search engine goes back to the previous position, where it could take a different path. And due to an inefficient regex pattern, it can create a long-running loop.

A pattern that can get stuck on carefully crafted input is called Evil Regex. Evil Regex contains Grouping with repetition and inside repeated group other repetitions and alternation with overlapping. Here are some examples:

- `(a+)+`
- `([a-zA-Z]+)*`
- `(a|a?)+`
- `(.*a){x} for x \ > 10`

All of the examples are susceptible to the input „aaaaaaaaaaaaaaaaaaaaaaaa“.

This extension would be part of the MFB analyzer and the implementation could check for example nested quantifiers “(a+)*” or help avoid ORs “(b|b)*” and report to the user, where in the code the Evil Regex is used, what is its category (grouping, nested, repeated), what is the criticality, and give a solution. [31]

9.3 Weak Hash Functions

The SHA-1, MD2, MD4, and MD5 are weak hashing algorithms. This extension should seek old and compromised algorithms and recommend their substitute. The algorithm SHA-1 is not a recommended algorithm for the hash password, signature verification, and other uses. PBKDF2 should be used to hash passwords for example. Or for SHA-1 algorithm it should give recommendation for SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 algorithms, depending on the implementation. [2]
Vulnerable code:

```
MessageDigest sha1Digest = MessageDigest.getInstance("SHA1");
    sha1Digest.update(password.getBytes());
    byte[] hashValue = sha1Digest.digest();
```

Example solution:

```
public static byte[] getEncryptedPassword(String password, byte[] salt)
throws NoSuchAlgorithmException, InvalidKeySpecException {
    KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 4096, 256 * 8);
    SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
    return f.generateSecret(spec).getEncoded();
}
```

The algorithms MD2, MD4, and MD5 are not recommended MessageDigest. PBKDF2 should be used to hash password for example or SHA-1 algorithm it should give recommendation for SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256.
Vulnerable code:

```
MessageDigest md5Digest = MessageDigest.getInstance("MD5");
    md5Digest.update(password.getBytes());
    byte[] hashValue = md5Digest.digest();
```

Example solution:

```
public static byte[] getEncryptedPassword(String password, byte[] salt)
throws NoSuchAlgorithmException, InvalidKeySpecException {
    KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 4096, 256 * 8);
    SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
    return f.generateSecret(spec).getEncoded();
}
```

9.4 XML Bomb Attack Checker

The XML bomb attack is a type of denial of service attack that targets applications or systems that process XML documents. The attack takes advantage of a feature in XML

called entity expansions, which allows an XML document to include references to external entities.

An attacker creates a small XML file that contains a reference to an external entity. This external entity, in turn, contains a reference to another external entity, and so on, creating a chain of nested entities that can quickly become very large. When the application or system processes the XML file, it attempts to expand these entities, which can lead to a significant increase in memory usage and CPU utilization. This can cause the application or system to slow down or crash, resulting in a denial of service. [16]
Here is an example of an XML Bomb:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
    <!ENTITY lol "lol">
    <!ELEMENT lolz (#PCDATA)>
    <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;">
    <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;">
    <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;">
    <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;">
    <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;">
    <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;">
    <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;">
    <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;">
    <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

There are a couple of ways of defending against this kind of attack. Here are some possibilities. Monitor resource usage. XML bomb attacks are designed to consume system resources, so monitoring resource usage can help detect when an attack is occurring. This can include monitoring CPU usage, memory usage, and network traffic. Limit entity expansion. Another way to prevent an XML bomb attack is to limit the number of nested entities that can be expanded. Many XML parsers have a limit on the number of nested entities they will expand by default, and this can be configured to limit the potential impact of an XML bomb attack.

9.5 XSS Via JSP Libraries Prevention

Cross-site scripting (XSS) is a type of security vulnerability in web applications that allows an attacker to inject malicious code (usually JavaScript) into a web page viewed by other users.

XSS attacks occur when an attacker is able to inject untrusted data, such as user input or data retrieved from a database, into a web page without proper input validation or output encoding. This can happen in various ways, such as through form fields, query parameters, or cookies.

When a victim user visits the compromised web page, the malicious code injected by the attacker is executed in the victim's browser, allowing the attacker to steal sensitive information, such as login credentials or personal data, or perform actions on behalf of the victim, such as sending unauthorized requests or modifying data.

JSP libraries provide a way to encapsulate and abstract complex functionality, making it easier for developers to create maintainable and reusable web applications. They can also be shared across multiple JSP pages or web applications.

However, if JSP libraries are not properly designed and implemented, they can be used as a vector for cross-site scripting (XSS) attacks. This is because JSP libraries can accept untrusted input, such as user input or data from a database, and include it in the output of the web page without proper input validation or output encoding.

If an attacker can inject malicious code into the untrusted input passed to a JSP library, that code can be executed in the browser of a user viewing the web page, allowing the attacker to steal sensitive information or perform actions on behalf of the victim. [17]

The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%...
 Statement stmt = conn.createStatement();
 ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
 if (rs != null) {
  rs.next();
  String name = rs.getString("name");
%>

Employee Name: <%= name %>
```

This code functions correctly when the values of the name are well-behaved, but it does nothing to prevent exploits if eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

9.6 Insecure Random Functions

The InsecureRandomness rule is a security best practice that detects the use of weak or predictable random number generators that can be exploited by attackers to compromise security. In cryptography, random number generators are used to generate keys, initialization vectors, and other security-critical parameters.

If a weak or predictable random number generator is used, an attacker can potentially predict or reproduce the security-critical parameters and use them to compromise the security of the system. [15]

Here is an example of an insecure Random function:

```
Random random = new Random();
int number = random.nextInt(10);
System.out.println("Random number: " + number);
```

Here is an example of a secure Random function:

```
SecureRandom random = new SecureRandom();
byte[] bytes = new byte[16];
random.nextBytes(bytes);
System.out.println("Random bytes: " + bytes);
```

The SecureRandom uses a strong algorithm, such as SHA-1, SHA-256, or SHA-512, to generate random numbers.

It also uses a random seed value to initialize the algorithm, which helps to ensure that the output of the algorithm is unpredictable. The seed value is typically generated from a variety of sources, such as hardware events, user input, and other sources of randomness.

SecureRandom is designed to generate random numbers that are non-deterministic, meaning that the output cannot be predicted based on previous outputs or knowledge of the algorithm. This makes it difficult for an attacker to reproduce the output of the algorithm.

Chapter 10

Conclusion

In conclusion, we have discussed the importance of code analysis, testing, and static analysis, and how they contribute to the development of efficient, reliable, and secure software. We have explored the history of software development and the challenges associated with creating bug-free software. We have described various techniques and tools for testing coded and discussed the advantages and disadvantages of black-box, gray-box, and white-box testing. We have also highlighted the significance of static analysis in identifying potential issues in code.

Furthermore, we have focused on the Eclipse platform and its plugins, and how they can be used to analyze code. We have examined the architecture of the tested system and the tool that we have integrated into the Eclipse platform as a plugin.

We have designed and implemented the extensions of the MFB Analyzer in particular, such as the Log4j checker, the GDPR extension, and the Eclipse marking extension. We have tested our research and implementations of the extensions and the marking plugin on dummy data and real data. The correct functionality was proven by the MFB Analyzer results. See the results in Appendix A.1, A.2, A.3. As we were testing the Extensions of the MFB Analyzer on real data, we have not found any threats, confirming that BOC-GROUP has clean and bug-free software and that these extensions will serve as prevention against bugs that could be added in the future.

At last, the possible extensions to the MFB Analyzer show the potential for further development and improvement in the field of code analysis and testing.

Bibliography

[1] AG, B. P. . S. *ADONIS Method and Metamodel* [online]. 2023. 20.1.2023. Available at: https://docs.boc-group.com/adonis/en/docs/15.0/user_manual/amm-000000/.

[2] Arteau, P. *Bugs Patterns* [online]. 20.1.2023. Available at: https://find-sec-bugs.github.io/bugs.htm.

[3] Bill Pugh, D. H. *FindBugs™ Fact Sheet* [online]. 2015. 20.1.2023. Available at: https://findbugs.sourceforge.net/factSheet.html.

[4] Chrisite, T. *False positives* [online]. 2014. 20.1.2023. Available at: https://codechecker.readthedocs.io/en/latest/analyzer/false_positives/.

[5] community spotbugs. *Bug descriptions* [online]. 2016-2022. 20.1.2023. Available at: https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html.

[6] Corp., I. B. M. *Eclipse Platform Technical Overview* [online]. 2006. 20.1.2023. Available at: https://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html.

[7] Corporation, I. *Dependency analysis* [online]. 2021. 20.1.2023. Available at: https://www.ibm.com/docs/en/wsr-and-r/8.5.6?topic=analysis-dependency.

[8] Darfler, B. *Top 5 Static Analysis Plugins for Eclipse* [online]. 2009. 20.1.2023. Available at: https://bdarfler.medium.com/top-5-static-analysis-plugins-for-eclipse-a73d944119af.

[9] Dynatrace. *What is Log4Shell?* December 2021. Available at: https://www.dynatrace.com/news/blog/what-is-log4shell/.

[10] Fahmy, A. Ariane 501 Rocket Explosion. [online]. Research Gate. april 2020, p. 7, 20.1.2023, [cit. 20.1.2023]. Available at: https://www.researchgate.net/publication/340875112_Ariane_501_Rocket_Explosion.

[11] Foundation, E. *Language Packs: 3.2* [online]. 2006. 20.1.2023. Available at: https://archive.eclipse.org/eclipse/downloads/drops/L-3.2_Language_Packs-200607121700/.

[12] Holzner, S. *Eclipse Cookbook*. O'Reilly Media, Incorporated, 2004. Cookbook Series. ISBN 9780596007102. Available at: https://books.google.cz/books?id=cGYMActRiakC.

[13] Johansson, S. *GDPR Compliance with Web Server Logs* [Ctrl Blog]. May 2018. 20.1.2023. Available at: https://www.ctrl.blog/entry/gdpr-web-server-logs.html.

[14] Loskutov, A. *JDepend4Eclipse* [online]. 2004. 20.1.2023. Available at: https://marketplace.eclipse.org/content/jdepend4eclipse.

[15] OWASP. *Insecure Randomness*. 2021. Available at: https://owasp.org/www-community/vulnerabilities/Insecure_Randomness.

[16] OWASP. *XML External Entity (XXE) Processing*. 2021. Available at: https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing.

[17] OWASP. *XSS (Cross Site Scripting) Prevention Cheat Sheet*. 2023. Available at: https://owasp.org/www-community/attacks/xss/.

[18] PMD. *UnusedPrivateMethodRule.java* [online]. 2021. 20.1.2023. Available at: https://github.com/pmd/pmd/blob/master/pmd-java/src/main/java/net/sourceforge/pmd/lang/java/rule/bestpractices/UnusedPrivateMethodRule.java.

[19] PMD. *Best Practices* [online]. 2022. 31.12.2022. Available at: https://pmd.github.io/latest/pmd_rules_java_bestpractices.html#unusedprivatemethod.

[20] Project, P. O. S. *Documentation Index* [online]. 2022. 20.1.2023. Available at: https://pmd.github.io/latest/.

[21] Project, P. O. S. *Tools / Integrations* [online]. 2022. 20.1.2023. Available at: https://pmd.github.io/latest/pmd_userdocs_tools.html.

[22] Setiawan, H., Erlangga, L. E. and Baskoro, I. Vulnerability Analysis Using The Interactive Application Security Testing (IAST) Approach For Government X Website Applications. In: *2020 3rd International Conference on Information and Communications Technology (ICOIACT)*. 2020, p. 471–475. DOI: 10.1109/ICOIACT50329.2020.9332116.

[23] S.M.K, Q. and Farooq, S. U. Software Testing – Goals, Principles, and Limitations. *International Journal of Computer Applications*. september 2010, vol. 6. DOI: 10.5120/1343-1448. Available at: https://www.researchgate.net/publication/46280097_Software_Testing_-_Goals_Principles_and_Limitations.

[24] snakile. *What are the differences between PMD and FindBugs?* [online]. 2010. 20.1.2023. Available at: https://stackoverflow.com/questions/4297014/what-are-the-differences-between-pmd-and-findbugs.

[25] snakile. *What are the differences between PMD and FindBugs?* [online]. 2010. 20.1.2023. Available at: https://stackoverflow.com/questions/4297014/what-are-the-differences-between-pmd-and-findbugs.

[26] Snyk. *Understanding gray box testing techniques* [online]. 2023. 19.1.2023. Available at: https://snyk.io/learn/application-security/testing/gray-box-testing/.

[27] Snyk. *White box testing basics: Identifying security risks early in the SDLC* [online]. 2023. 19.1.2023. Available at: https://snyk.io/learn/application-security/testing/white-box-testing/.

[28] Team, C. *CodeMR* [online]. 2023. 20.1.2023. Available at: https://marketplace.eclipse.org/content/codemr-static-code-analyser.

[29] Team, S. *Spotbugs* [online]. 20.1.2023. Available at: https://spotbugs.github.io/.

[30] Union, E. *General Data Protection Regulation (GDPR)*. April 2016. Official Journal of the European Union, L 119/1. Available at: https://gdpr-info.eu/.

[31] Weidman, A. *Regular expression Denial of Service - ReDoS* [online]. 2023. 20.1.2023. Available at: https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS.

[32] Wikipedia. *ADONIS (software)* [online]. 2022. 20.1.2023. Available at: https://en.wikipedia.org/wiki/ADONIS_(software).

[33] Wikipedia. *FindBugs* [online]. 2022. 20.1.2023. Available at: https://en.wikipedia.org/wiki/FindBugs.

[34] Wikipedia. *Mars Climate Orbiter* [online]. 2023. 20.1.2023. Available at: https://en.wikipedia.org/wiki/Mars_Climate_Orbiter.

[35] Wikipedia. *Morris worm* [online]. 2023. 20.1.2023. Available at: https://en.wikipedia.org/wiki/Morris_worm.

[36] Xiaofeng, T. *CVE-2021-44228-Apache-Log4j-Rce*. December 2021. Commit hash: 321ab5c. Available at: https://github.com/tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce.

Appendix A

MFB Analyzer Results

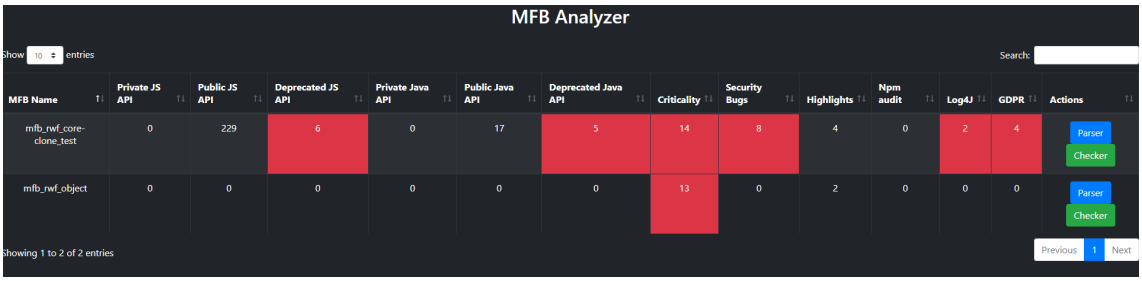


Figure A.1: MFB Analyzer Results Summary

Figure A.2: MFB Analyzer Results Log4j

Figure A.3: MFB Analyzer Results GDPR