**Gottfried Wilhelm
Leibniz Universität Hannover
Faculty of Electrical Engineering and Computer Science
Institute of Practical Computer Science
Software Engineering Group**

Advancement of Security Monitoring in JIRA with JavaScript Support

Bachelor Thesis

in Computer Science

by

Amin Akbariazirani

**First Examiner: Prof. Dr. Kurt Schneider
Second Examiner: Dr. Jil Klünder
Supervisor: M. Sc. Fabien Patrick Viertel**

Hannover, August 26, 2019

Declaration of Independence

I hereby certify that I have written the present bachelor thesis independently and without outside help and that I have not used any sources and aids other than those specified in the work. The work has not yet been submitted to any other examination office in the same or similar form.

Hannover, August 26, 2019

Amin Akbariazirani

Abstract

The dependence of human activities on software is increasing by day. Although the benefits of using computers are undoubtedly significant, there may also be some risks involved. This includes the possibility of software vulnerabilities potentially resulting in a breach of the predefined security policy. This places a heavy burden on software developers to create software artifacts that are as free as possible from potential exploits due to vulnerabilities. In a software project, both source code and the imported libraries can contain vulnerabilities. Manual analysis of software projects in search for vulnerabilities is not only time consuming but also requires corresponding knowledge. To enhance this process, software solutions such as the ProDynamics plugin for the JIRA project management software can be used.

The ProDynamics plug-in, which was developed to assist teams in various phases of software development including security assurance, is equipped with a vulnerability detector for the programming language Java. The vulnerability detector analyzes sprint cycles and provides software developers with information about possible vulnerabilities both in the source code and in the used libraries.

In course of this thesis, the ProDynamics plugin, in particular the "Security Checker" module, is extended to also support JavaScript. This process includes developing a JavaScript tokenizer and defining code blocks according to the specifications of the JavaScript syntax. Since JavaScript source code and library files share the same file extension, ".js", a case by case analysis of JavaScript files might not be sufficient for accurate library analysis. The library checker is therefore extended to not only process files based on the broad characteristics of JavaScript library files but also to extract libraries which are imported through the package manager NPM.

Contents

Chapter 1

Introduction

Software has directly or indirectly become a part of our day to day lives. From buying groceries to our mobile phones, software has probably played a part in either the process of purchasing, manufacturing or shipping of the product. As time goes by, human activities are becoming more and more dependant on computers. This has led to the number of microcontrollers and microprocessors largely surpassing the total human population [9]. This puts an enormous burden on software developers to create software that is as free as possible from all sorts of vulnerabilities. With software infiltrating many aspects of our lives, the crucial task of finding and neutralizing software vulnerabilities is gaining in importance.

Articles about data breaches and exploited software vulnerabilities are commonplace in the news. A very recent example is the discovery of a targeted surveillance attack on the messaging app WhatsApp where a software vulnerability was used by hackers to read “end-to-end encrypted” massages exchanged on the platform [16].

The task of identifying and fixing possible software vulnerabilities is of utmost importance. Vulnerabilities can either be contained in the source code or in the included libraries. In order to identify all possible vulnerabilities, a proper approach would be to identify third-party libraries and check them against databases like the National Vulnerability Database (NVD) [29] and also to compare the source code itself to a database of already known vulnerable code [28].

1.1 Problem

Since the process of finding vulnerabilities is not only time consuming but also requires a certain set of skills [2], a more efficient use of resources is to semi-automate this process and to outsource this crucial but knowledge intensive task to computers. In recent years, JavaScript has evolved into one of the most popular programming languages. In addition to client-side web applications, it is also used for server-side, mobile and desktop applications [27]. Methods which are used for vulnerability detection can under circumstances be shared across different programming languages [2] but since code blocks and syntaxes can vary, they often need to be analyzed separately. Hence, an independent research has to be done in order to efficiently detect JavaScript vulnerabilities.

1.2 Proposed Solution

A plugin developed by M. Matthaei for the project management software JIRA is the basis of this thesis [20]. This plugin is designed to automate the process of finding vulnerabilities for projects written in the programming language Java. This is done by comparing code blocks and libraries with corresponding vulnerability databases. The purpose of this thesis is to extend the above mentioned plugin to support JavaScript. This was achieved by analyzing the project to identify JavaScript libraries and their associated metadata, and by implementing an approach to detect code clones under JavaScript.

1.3 Structure of the Thesis

This thesis is structured as follows. In chapter two, the key concepts which are used in this thesis will be discussed. This will include some basics about vulnerabilities, the used vulnerability databases and the programming language JavaScript. To help understand this project better, the subsequent chapter is dedicated to introducing some related works to this thesis. In chapter four the concepts of code clone detection and library analysis will be presented. The implementation of these concepts and the unexpected hardships which appeared along the way will be discussed in chapter five. Chapter six will be focusing on evaluating the implemented vulnerability detector for the programming language JavaScript. The thesis will be wrapped up in chapter eight and possible ideas for further development of the software will be proposed.

Chapter 2

Basics

In this chapter, the fundamental concepts and tools which were used for this project will be presented and discussed. This includes but is not limited to defining software vulnerability, introducing the principles of code clone detection, elaborating possible approaches for vulnerability detection in libraries and explaining some characteristics of the programming language JavaScript.

2.1 Software Vulnerability

According to I. V. Krsul, "a software vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy" [15]. According to the MITRE Corporation and with focus on the source code of a software product, the definition of a vulnerability can be defined as a weakness in the code that with its exploitation might result in a negative impact to confidentiality, integrity, or availability [21]. As the mentioned definitions point out, vulnerabilities in the source code might affect information security so it is an important task to identify and fix them in order to ensure information security.

2.2 Code Clone Detection

Identifying vulnerable code is not a task typical software engineers are trained to fulfill [28]. Fortunately, there are publicly accessible vulnerability databases that can be used to facilitate this task. For instance, the National Vulnerability Database aka NVD. This database consists of known vulnerabilities and some corresponding information about them. This includes among other things, a base score identifying the criticality of the vulnerability, some references to advisories, solutions and tools and a Common Vulnerabilities and Exposures ID (CVE ID) which can be used to classify vulnerabilities accordingly. Though the NVD database enlists many relevant information, code samples are rarely included [28]. The availability of a vulnerable code database is a vital prerequisite for this project and the approach used in [28] for extracting vulnerable code samples from large scale repositories like GitHub has been used for this purpose.

In order to find similarities between source code and known vulnerable code snippets, the possibility of diversified syntaxing has to be taken into consideration. In this project, the concept called SourcererCC introduced in [25] has been adopted to tackle the above challenge. This approach, which makes use of the tokenization of code snippets, will help us identify many code clones but comes at the cost of a certain amount of similarity in syntax, being crucial for it to function properly. Figure 2.1 shows the process of code clone detection as used in this project and implemented by [28].

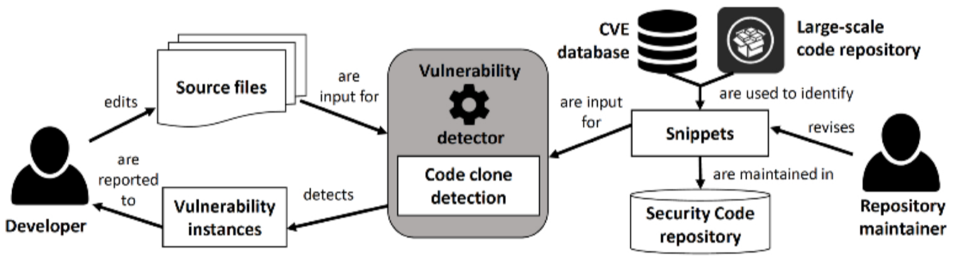


Figure 2.1: Code clone detection as implemented in [28]

2.3 Detecting Vulnerable Libraries

In the world of software development, reusing code through available libraries is done quite regularly [29]. Research has shown that reusing already available software artifacts increases the productivity of a development team, speeds up the time-to-market process and improves the quality of the resulting software [10, 17]. On the other hand the more libraries are used, the greater the probability of exposure to vulnerabilities may get [30]. It is therefore good practice to identify vulnerable libraries at an early stage. In order to do so, an effective approach would be to collect the metadata of all libraries and check them against a database of known vulnerable libraries. As discussed in 2.2, every entry in the NVD database consists of different information, a few of which have been mentioned earlier.

Another useful information available in the NVD entries is the Common Platform Enumeration (CPE) which consists of different information about the products in which the corresponding vulnerability has been detected in. This information includes but is not limited to the product's name, vendor and version number. The convenient structure of the CPE can therefore be used to efficiently search for entries in the NVD database which correspond to the metadata of a library. The above approach has been implemented

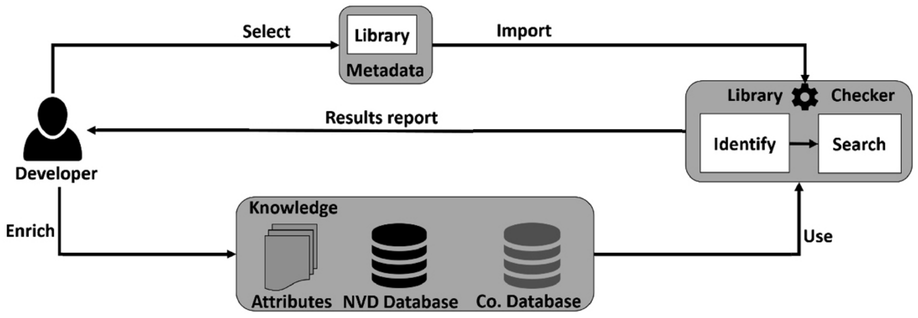


Figure 2.2: Library detection as implemented in [29]

in [29] for the programming language Java. This thesis uses the mentioned software as its basis for the JavaScript library checker. Figure 2.2 shows the process of how the library checker is originally used to identify vulnerable libraries as implemented by [29].

2.4 JavaScript

The programming language JavaScript was first introduced by the Netscape Communications Corporation in 1995. According to the initial press released by the Netscape corporation, "JavaScript is an easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers“ [3]. This was the initial idea behind JavaScript. Today, JavaScript can be used for both front-end an back-end development. In order to standardize scripting languages like JavaScript, Ecma International created the ECMAScript scripting language specification defined in *ECMA-262* and *ISO/IEC 16262*. Since the initial release of ECMAScript in 1997, 9 new editions have been published [4]. Many containing new syntax and semantic support. Table 2.1 summarizes the changes to ECMAScript over time. This gradual updates is one of the points that has to be kept in mind while developing software aimed at the programming language JavaScript.

2.5 Node Package Manager (NPM)

One of the cornerstones of efficient software development is to reuse software artifacts developed by other programmers. In order to support developers, modern software ecosystems often include package managers to automate the process of installing and maintaining software dependencies. One the most well known package managers for the programming language JavaScript is the Node Package Manager short NPM. As of February 2019, NPM hosts 800,000 software packages and it is expected that this number will continue

Ver.	Official Name	Description
1	ECMAScript 1	First Edition.
2	ECMAScript 2	Editorial changes only.
3	ECMAScript 3	Added Regular Expressions. Added try/catch.
5	ECMAScript 5	Added ”strict mode“. Added JSON support. Added String.trim(). Added Array.isArray(). Added Array Iteration Methods.
5.1	ECMAScript 5.1	Editorial changes.
6	ECMAScript 2015	Added let and const. Added default parameter values. Added Array.find(). Added Array.findIndex().
7	ECMAScript 2016	Added exponential operator (**). Added Array.prototype.includes.
8	ECMAScript 2017	Added string padding. Added new Object properties. Added Async functions. Added Shared Memory.
9	ECMAScript 2018	Added rest/spread properties. Added Asynchronous iteration. Added Promise.finally(). Additions to RegExp.

Table 2.1: ECMAScript changes over time [4]

to grow. As a major source of JavaScript packages for client and server side projects, NPM has become one of the central component of JavaScript development [31].

2.6 JIRA

JIRA is a web based project management software developed by Atlassian. It can be used for issue tracking and project management. According to Atlassian, JIRA is used by 150,000+ teams worldwide [1]. One of the customers of JIRA, are software developers who can use it to manage scrum cycles and improve the development process accordingly. According to the official Scrum Body of Knowledge, ”Scrum is founded on empirical process control theory, or empiricism. Empiricism asserts that knowledge comes from experience and making decisions based on what is known. Scrum employs an iterative, incremental approach to optimize predictability and control risk“ [26]. The key concept in the scrum framework is the iterative approach which is defined by time cycles in which participants have to fulfill specific assignments. JIRA also offers a plugin system which can be used to develop independent plugins adding features to it.

Chapter 3

Related Works

According to W. Jimenez et al., detecting software vulnerabilities in the process of software construction can be performed by tools that are either designed based on static and/or dynamic techniques [11]. Static techniques refer to tools which analyze the source code without executing it. On the other hand, dynamic techniques are tools that run the source code in a controlled environment and collect relevant information for vulnerability detection. The approach implemented in this thesis which is a continuation to W. Brunotte's master thesis, [2] uses static techniques, in particular lexical analysis, to detect vulnerabilities in the source code.

The same paper states that in lexical analysis, a sequence of tokens is extracted from the source code and compared with a database of known vulnerabilities. "SourcererCC“, the underlying software artifact used in this thesis for clone detection was briefly described in section 2.2. Although "SourcererCC“ was designed to be reliable, complete, fast and scalable [25], W. Jimenez et al. state that tools created based on lexical analysis have generally the downside of generating a high number of false positives due to the non-observance of syntax and grammar. The proposed approach by W. Jimenez et al. is to create vulnerability models that are based on particular conditions of the source code. If any of the condition occur, the related vulnerability will be detected.

S. Kals et al. sub-classifies the procedure of testing applications for the presence of bugs and vulnerabilities into two main categories. Black box and white box testing. A procedure is referred to as white box testing, if the vulnerability detection process relies on scanning the source code for possible deficiencies. On the other hand in black box testing, The source code is not available for the test environment and special uses cases are sent to a running instance of the software. A comparison between the returned values and the expected output, will determine the presence of possible vulnerabilities [12]. In the process of this thesis the aim was to create a JavaScript vulnerability detection method according to the concept of white box testing.

The programming language JavaScript is widely used in web development thus, the black-box web vulnerability scanner "SecuBat" which was developed by S. Kals et al. can be a complementary black box testing tool for the JavaScript vulnerability detector developed in the course of this thesis. According to S. Kals et al., "SecuBat is a generic and modular web vulnerability scanner that, similar to a port scanner, automatically analyzes web sites with the aim of finding exploitable SQL injection and XSS vulnerabilities." [12].

Chapter 4

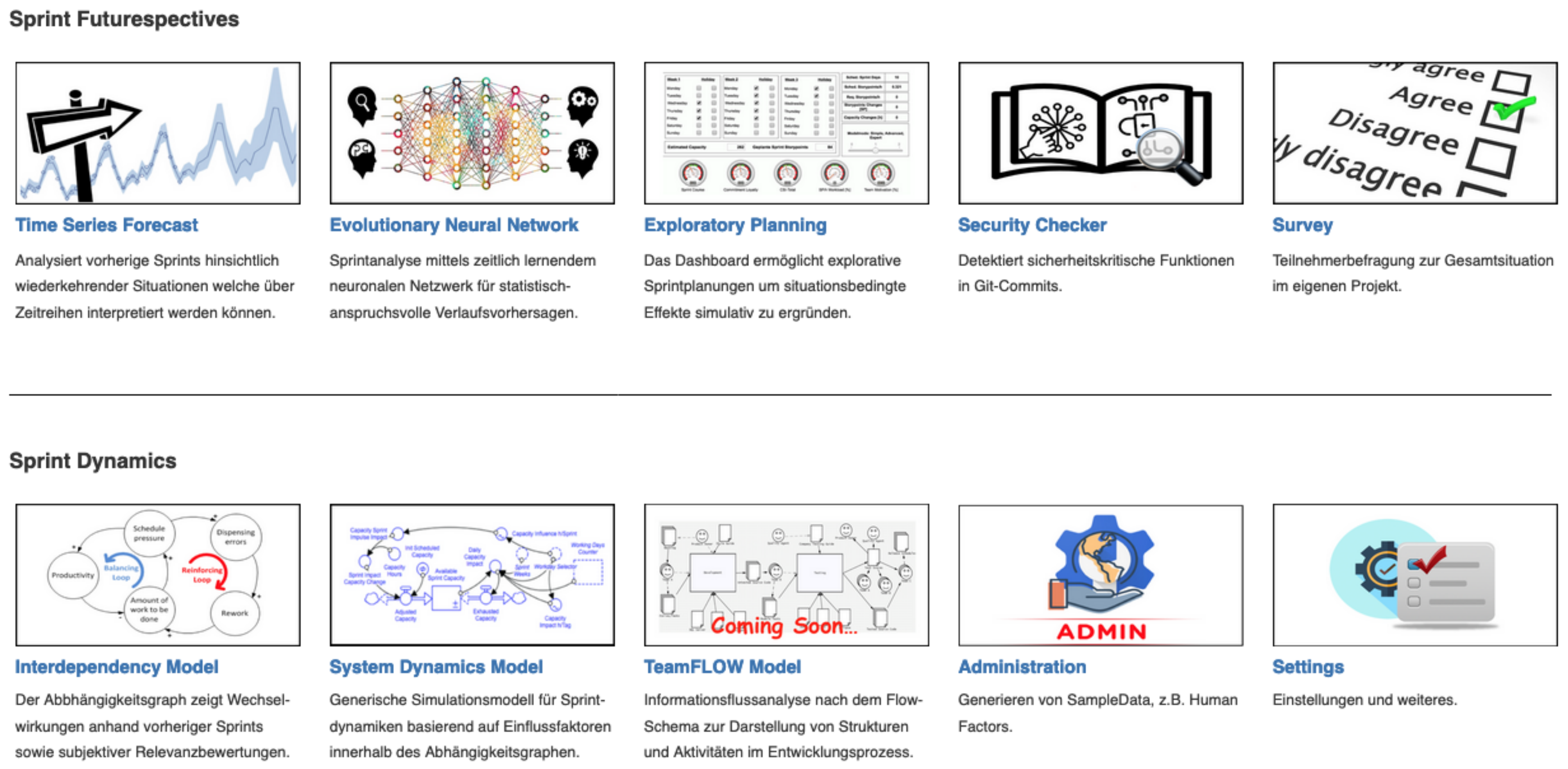
Concept

The goal of this thesis is to extend the already available vulnerability detection software for Java projects defined in 2.2 and 2.3 to support JavaScript. Therefore, it is necessary to understand the structure of the JavaScript development ecosystem and the most used tools regarding this programming language. In this chapter, the underlying JIRA plugin for which this project was developed will be introduced, the programming language JavaScript will be examined in more detail and the relevant differences between JavaScript and Java will be discussed.

4.1 ProDynamics Plugin for JIRA

ProDynamics is a plugin developed for the project management software JIRA. It can be used to analyze and manage various aspects of a software project with regard to each development sprint. Using ProDynamics, a project can be examined in its retrospective, futurespective and dynamic aspects. This project, as a continuation to the security checker plugin, is developed as part of the futurespective functionality of the ProDynamics JIRA plugin. Figure 4.1 shows the overview page of the ProDynamics plugin which was introduced by F. Kortum et al. in [14].

This thesis is a continuation of the ”Security Checker“ functionality developed by M. Matthaei [20]. The architecture of the ”Security Checker“ has been adapted and customized to support both code and library checking functionalities in Java and JavaScript. Figure 4.2 offers and overview of the architecture behind the ”Security Checker“ as implemented in course of this thesis. The architecture is described more closely in chapter 5.



Sprint Futurespectives

Time Series Forecast
Analysiert vorherige Sprints hinsichtlich wiederkehrender Situationen welche über Zeitreihen interpretiert werden können.

Evolutionary Neural Network
Sprintanalyse mittels zeitlich lernendem neuronalen Netzwerk für statistisch-anspruchsvolle Verlaufsvorhersagen.

Exploratory Planning
Das Dashboard ermöglicht explorative Sprintplanungen um situationsbedingte Effekte simulativ zu ergründen.

Security Checker
Detektiert sicherheitskritische Funktionen in Git-Commits.

Survey
Teilnehmerbefragung zur Gesamtsituation im eigenen Projekt.

Sprint Dynamics

Interdependency Model
Der Abbhängigkeitsgraph zeigt Wechsel-wirkungen anhand vorheriger Sprints sowie subjektiver Relevanzbewertungen.

System Dynamics Model
Generische Simulationsmodell für Sprint-dynamiken basierend auf Einflussfaktoren innerhalb des Abhängigkeitsgraphen.

TeamFLOW Model
Informationsflussanalyse nach dem Flow-Schema zur Darstellung von Strukturen und Aktivitäten im Entwicklungsprozess.

Administration
Generieren von SampleData, z.B. Human Factors.

Settings
Einstellungen und weiteres.

Figure 4.1: Overview of the ProDynamics JIRA plugin [14]

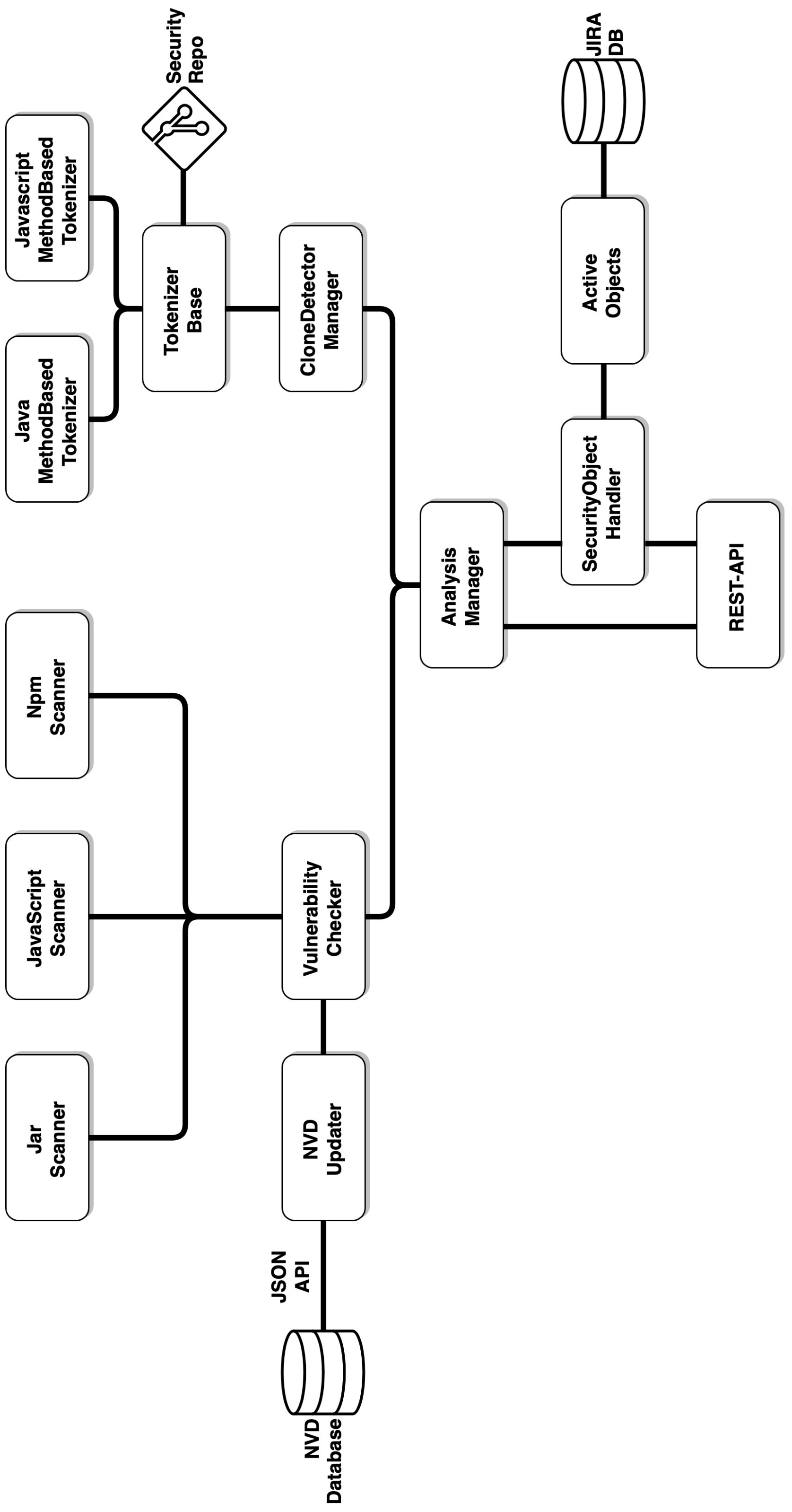


Figure 4.2: Architecture Diagram of the "Security Checker"

4.2 Detecting Source Code Vulnerabilities

According to [28] which was used as the basis for this thesis's JavaScript code clone detection, the process of finding code clones consists of the following four steps:

1. Pre-Processing
2. Code Processing
3. Clone Detection
4. Generating Results

In steps one and two, code blocks extracted from the vulnerability database and the to be processed source code files have to be parsed, tokenized and indexed. For this purpose, an inverted index is created for each code block, containing its respective tokens and the times each token was used inside the code block. The list of all inverted indexes is then saved into the book keeping information file which keeps track of all files and, if available, its corresponding CVE Id. In step three, "SourcererCC" will use the above information to match code blocks from the source code to code blocks extracted from the vulnerability database and saves the discovered code clones to the database.

4.2.1 The Evolution of JavaScript

In 2.4 it was pointed out that the programming language JavaScript has been standardized by the ECMAScript scripting language specification. This in turn has created different versions of JavaScript which are backward compatible and diverse in syntax. As an example, in ECMAScript 2015, the keywords "let" and "const" where added to the list of keywords supported by JavaScript. These keywords are used to declare variables alongside the original "def". In JavaScript, variables can also be declared without a prefix keyword. The diverse syntax of JavaScript has to be taken note of in the process of parsing.

One of the main differences between Java and JavaScript is that JavaScript has a more relaxed syntax and fewer rules. An example for this would be that in Java, code needs to be written inside classes, and functions play a fundamental role in delivering logic to the code. On the other hand in JavaScript code and logic can be written anywhere. A very obvious instance of the above is the main function in Java as the execution point of the whole project. In JavaScript the code itself is executed at run-time and a main function is basically irrelevant unless coded accordingly.

Prior to ECMAScript 6 (ES6), JavaScript was considered a solely prototype-based language (inheritance with code reuse) but with the release of ES6, JavaScript has also adopted the class-based nature of Java. So JavaScript now supports both approaches. Listing 4.1 shows how a class is defined in Java.

```java
public class Sample {
    String name;
    Sample(String name){
        this.name = name;
    }
    public static void main(String args[]) {
      Sample block1 = new Sample("This is a sample class");
      System.out.println(block1.name);
      // Prints "This is a sample class"
    }
}
```

Listing 4.1: Defining and using a class in Java

Listing 4.2 is the same Java class defined in JavaScript version ES6.

```javascript
class Sample {
    constructor(name) {
        this.name = name;
    }
}
const block1 = new Sample('This is a sample class');
console.log(block1.name);
// Prints "This is a sample class"
```

Listing 4.2: Defining and using a class in JavaScript

Prior to the release of ES6, the class syntax was not available therefore a JavaScript object had to be defined differently. Listing 4.3 shows how the same Java class has to be defined in JavaScript prior to the release of version ES6.

```javascript
var block1 = {
    name : 'This is a sample class'
};
console.log(block1.name);
// Prints "This is a sample class"
```

Listing 4.3: Objects in JavaScript prior to ECMAScript 6 (ES6)

4.2.2 Defining Blocks

The code clone detector which was first mentioned in 2.2, is one of the key basics of this thesis. This approach, first introduced by H.Sanjani with the name ”SourcererCC“, uses code blocks in order to identify similarities between two code snippets [25]. These blocks have to be identified according to the programming language which is being processed. For the programming language Java, F. P. Viertel et al. decided to use methods and constructors as blocks [28]. Since the programming languages JavaScript and Java have a few differences in terms of Syntax, code blocks need to be defined differently. One of the main differences between the two is that in JavaScript it is common practice to write code outside of functions. This means that a JavaScript file might consist of multiple lines of code but not include any functions. So code written outside of functions have to be identified as code blocks. Since code blocks might be written anywhere outside of functions, lines which are written in between functions can also be singled out as code blocks.

Another characteristic of JavaScript is that functions can be defined as variables. All function may in turn contain more functions resulting in a nested list of functions within each other. Functions can also return functions as output. These inner functions are called "closures". Closures are one of the key capabilities of JavaScript. These nested functions have access to all variables and functions of the outer function while not having access to the variables of their inner functions. This means that some variables and functions might have a longer lifetime than the scope in which they are defined in. A closure occurs when the inner function is made available to any scope outside the outer function [22]. See 4.4. This characteristic of JavaScript means that every inner function also has to be identified as a code block.

```
var func1 = function(bb){
  var func2 = function(vv){
    return function(cc){
        return vv + cc;
    };
  }
  return func2(bb);
}
console.log(func1(5)(10));
// Prints 15
```

Listing 4.4: Nested functions in JavaScript with a function as return value

Before 2015, JavaScript didn't support Java like objects in the form of classes but as of ECMAScript 6 classes are also part of JavaScript's syntax and semantic. See section 4.2.1. Classes can not only define an object but also contain logic. This means that a class also has the capacity to be defined as a block.

4.3 Detecting Library Vulnerabilities

In addition to the source code, libraries can also contain vulnerabilities. So it of utmost importance to not only scan the source code for vulnerabilities but also libraries. In JavaScript, libraries have the same file extension as source code. So libraries have to be identified as such accordingly.

4.3.1 Identifying NPM Libraries

As mentioned in section 2.5, node package manager (NPM) is a major source for JavaScript libraries for client and server side projects. This assumption can be validated by analyzing some of the trending JavaScript repositories on GitHub. Table 4.1 lists 20 trending JavaScript GitHub libraries and shows how widespread the use of NPM is. In the examined libraries, 14 out of 20 use NPM as their desired package manager.

Repo Name on GitHub	**Uses NPM**
gothinkster / realworld	✔
algorithm-visualizer / algorithm-visualizer	✔
pixijs / pixi.js	✔
jaywcjlove / awesome-mac	✔
vuejs / vue	✔
bannedbook / fanqiang	
strapi / strapi	✔
azl397985856 / leetcode	
elastic / kibana	✔
mui-org / material-ui	✔
haotian-wang / google-access-helper	
facebook / react-native	✔
hasura / graphql-engine	
yangshun / tech-interview-handbook	
gatsbyjs / gatsby	✔
GoogleChrome / puppeteer	✔
carbon-design-system / carbon	✔
trazyn / ieaseMusic	✔
syhyz1990 / baiduyun	
axios / axios	✔

Table 4.1: The use of NPM in the 20 trending Javascript repos on GitHub. Stand June 2019 [6]

NPM is a major package manager for JavaScript and as shown in table 4.1, is widely used in JavaScript development. Projects which use

NPM, include files which identify the dependencies used in the project. The specified libraries will then be downloaded from the NPM server and included in the project accordingly. The file containing this information, is by default "package.json" and/or "package-lock.json" and is written in the Json file format.

The "package.json" and "package-lock.json"files contain among other things, the required dependencies for the project with the minimal versions and the exact versions respectively. Using these files for library identification has two significant advantages. First and foremost, the libraries which are used in the project are referenced in full. This means that the library names do not have to be guessed from the file name etc. On the other hand, the actual imported version of the libraries is also listed in the json files. Using precise library names and correct library versions, the process of finding vulnerabilities using Common Platform Enumeration (CPEs) results in an accurate list of entries from the National Vulnerability Database (NVD).

4.3.2 Identifying JavaScript Libraries

In the programming language Java, library files are saved as packages with the extension ".jar". A jar file contains a manifest file from which the library's name and version, if available, can be obtained. In JavaScript, library files are pure JavaScript code and have the same ".js" extension as other JavaScript files. This means that in contrast to Java libraries, JavaScript libraries have to be identified as such. This also means that meta information is not directly available. Since JavaScript libraries can exist anywhere inside a project, all ".js" files have to be processed individually.

In order to distinguish ".js" files from ".js" libraries, it is necessary to find a possible differentiating factor between them. In order to find this characteristic, the most starred GitHub JavaScript libraries with more than 40000 stars where examined (Stand July 2019). 16 of those repositories where JavaScript libraries. Table 4.2 lists those libraries and identifies if and where in the main ".js" file, relevant information is noted.

As shown in table 4.2, it is common in JavaScript development to add the version number to the first block comment of the main ".js" file. Hence, this is a good criteria for JavaScript library detection. Likewise, the library name is often noted in the file name of JavaScript libraries. The library name and the version number will then be used to find vulnerabilities using the Common Platform Enumeration (CPE). ".js" files which do not include a version number in the first block comment will finally be processed as JavaScript code.

Name	Lib Name in File Name	Version in File Name	Version in Comments	File Name	Version in Comment
vuejs/vue	✔		✔	vue.js	v2.6.10
twbs/bootstrap	✔		✔	bootstrap.min.js	v4.3.1
facebook/react	✔		✔	react.development.js	v16.9.0
d3/d3	✔	✔	✔	d3.v5.min.js	v5.9.7
facebook/react-native	✔		✔	react.development.js	v16.8.6
axios/axios	✔		✔	axios.min.js	v0.19.0
FortAwesome/Font-Awesome			✔	all.js	5.9.0
angular/angular.js	✔		✔	angular.min.js	v1.7.8
mrdoob/three.js	✔			three.min.js	
jquery/jquery	✔	✔	✔	jquery-3.4.1.min.js	v3.4.1
mui-org/material-ui	✔		✔	material-ui.production.min.js	v4.2.0
hakimel/reveal.js	✔			reveal.js	
socketio/socket.io	✔		✔	socket.io.js	v2.2.0
Semantic-Org/Semantic-UI	✔		✔	semantic.js	2.4.1
chartjs/Chart.js	✔		✔	Chart.min.js	v2.8.0
moment/moment	✔			moment.js	

Table 4.2: Analysis of the main ".js" file of most starred JavaScript libraries on GitHub. Stand July 2019 [5]

Chapter 5

Implementation

In this chapter, the implementation of the JavaScript code clone detector and the library checker will be explained. Furthermore, general improvements to the "Security Checker" including but not limited to support for the JSON interface of the NVD API will be discussed.

5.1 Source Code Vulnerability Detector

Finding code clones depends on pre-processing of the "Security Repo". In the process of initializing the code clone detector, the "Security Repo" which includes code snippets with known vulnerabilities is tokenized and the results are saved to the file storage. When code clone detection is run through the "Analysis Manager", ".js" source code files which are according to section 5.2 not libraries, are tokenized and compared to the previously tokenized files inside the "Security Repo".

The results are then handed over to the "AnalysisManager" and saved to the JIRA database. JavaScript parsing and tokenizing which is relevant to both initialization and execution of the code clone detector are explained in more detail in the following section.

5.1.1 JavaScript Parser and Tokenizer

"SourcererCC" is equipped with a built-in Java tokenizer which was used by the original code clone detector of ProDynamics. For JavaScript, a corresponding tokenizer was implemented. For this purpose, the library "Closure-Compiler" by Google was used. This library's original use is to improve JavaScript's execution efficiency by parsing, analyzing, removing dead code, rewriting and minimizing JavaScript source code [7]. This library which is also available in the Maven repository, includes a JavaScript parser. The parser creates a node tree from the source code which can be traversed and processed according to the developer's specific need.

As discussed in 4.2.2, defining code blocks for code clone detection using ”SourcererCC“ is a fundamental part of the process. In the above mentioned section, code block basics for JavaScript have been reviewed comprehensively. A JavaScript code block is defined in this project as a Java class called ”JsBlock“ which keeps track of the list of tokens, the start and end line numbers of the code block, the identifying name of the block and a list of the internally processed nodes.

A major advantage of using ”Closure Compiler“ for JavaScript parsing is the fact that the created nodes can be traversed independently for sub nodes. This feature is used to create internally processed, independent block objects. The process of tokenizing starts from the ”AnalysisManager“ class. A call for the ”CloneDetectorManager“ is created which will in turn trigger ”TokenizerBase“ to instantiated the corresponding tokenizer for Java (JavaMethodBasedTokenizer) and JavaScript (JavaScriptMethodBasedTokenizer) files respectively. Identified blocks are then processed accordingly and the corresponding token information is saved to the file storage. The information about each block is saved as ”Book Keeping Information“ and the tokens are saved in a separate file. The realization of tokenization as implemented in W. Brunotte's masters thesis [2] is as follows.

```
var num1 = 10;
var num2 = 20;
var sum = num1+num2;
var text = "The sum of" + num1 + " and " + num2 + " is " + sum;
console.log(text);
```

Listing 5.1: Sample JavaScript code

Consider listing 5.1. The tokenizer extracts the tokens from the corresponding block, which in this case is only one, and saves it to the tokens file. The tokens are saved as shown in listing 5.2. The data can be divided in two parts by ”@#@“. The second part of the generate tokens data follows the schema <Token>@@::@@<Quantity>. The first part consists of a few information about the generated tokens and a few reserved values. The detailed info about the values is listed in table 5.1

```
54,54,1,14,0,0,0,0,0,13@#@log@@::@@1,sum@@::@@3,The@@::@@1,
    and@@::@@1,20.0@@::@@1,of@@::@@1,num1@@::@@3,text@@::@@2,
    num2@@::@@3,console@@::@@1,var@@::@@4,is@@::@@1,10.0@@::@@1
```

Listing 5.2: Tokens generated from listing 5.1

The generated tokens are extracted from a block inside a source code file. Additionally, in the initialization process where known vulnerable code snippets are tokenized, the corresponding CVE Id of the vulnerability also has to be saved. All the above information is saved to the ”Book Keeping Information“ file. Listing 5.3 shows the book keeping data generated for listing 5.1.

	Value	Description
1	Block ID	Block Identifier
2	Unique Tokenization ID	Method Identifier
3	Project ID	Project Identifier
4	Unique Resource ID	Code Resource Identifier
5	Reserved	Reserved for Future Impl.
6	Reserved	Reserved for Future Impl.
7	Reserved	Reserved for Future Impl.
8	Reserved	Reserved for Future Impl.
9	Reserved	Reserved for Future Impl.
10	Total Tokens	Total Number of Tokens

Table 5.1: Information saved in the first part of a token entry

```
54,14,54,4,8,0,2,0,0,0,0,0,1,CVE-0000-0000,1,9.3,/
    path_to_js_file/filename.js,Block_0
```

Listing 5.3: "Book Keeping Information" generated from listing 5.1

A "Book Keeping Information" entry consists of various information about a processed code block. This information is listed in table 5.2.

	Value	Description
1	Block ID	Block Identifier
2	Unique Resource ID	Code Resource Identifier
3	Unique Tokenization ID	Method Identifier
4	Start	Start Line Number
5	End	End Line Number
6	Granularity Level	Granularity Level Setting
7	Language	Corresponding Language
8	Resource Type	Type of Resource
9	Reserved	Reserved for Future Impl.
10	Reserved	Reserved for Future Impl.
11	Reserved	Reserved for Future Impl.
12	File ID	ID of the Vuln. File

13	Vulnerable ID	ID of the Vuln.
14	CVE ID	CVE ID of the Vuln.
15	Typ	Type of the Vuln.
16	Severity Score	Severity Score of the Vuln.
17	Resource Name	Name of the Resource
18	Name	Name of the Block

Table 5.2: Data saved in ”Book Keeping Information“ entries

5.2 Library Checker

In order to check libraries for vulnerabilities, the ”Security Checker“ plugin uses the approach explained in section 2.3. This approach makes use of the Common Platform Enumeration (CPE) to identify vulnerable libraries. The CPE entries are extracted from the NVD database through their XML or the newer JSON interfaces. In course of this project, the parser has been customized to support the JSON interface for data extraction from the NVD database, as support for the XML interface will end on October 9th, 2019. The process will be explained in section 5.4.

For library checking to work properly, specific information has to be extracted and passed to the ”VulnerabilityChecker“ class. For Java, the included manifest file is used to extract relevant information about the library of the included ”.jar“ file. In addition to the manifest file, the file name of a library might also be used for library name and version number identification. This process is done in the ”JarScanner“ class. For the programming language JavaScript, the ”VulnerabilityChecker“ receives data from either the ”NpmScanner“ or the ”JavaScriptScanner“. The ”NpmScanner“ processes library data received through the NPM packages and the ”JavaScriptScanner“, processes ”.js“ files. The ”NpmScanner“ searches the project for ”package.json“ and ”package-lock.json“ files and analyzes the files for dependencies. Listing 5.4 is a sample on how dependencies are stored in the ”package-lock.json“ file.

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
    "minimist": {
      "version": "0.0.10",
      "resolved": "https://registry.npmjs.org/minimist/-/
    minimist-0.0.10.tgz",
      "integrity": "sha1-3j+YVD2/lggr5IrRoMfNqDYwHc8="
    },
    "wordwrap": {
      "version": "0.0.3",
      "resolved": "https://registry.npmjs.org/wordwrap/-/
    wordwrap-0.0.3.tgz",
      "integrity": "sha1-o9XabNXAvAAI03I0u68b7WMFkQc="
    }
  }
}
```

Listing 5.4: A sample package-lock.json file

In the "package.json" file, dependencies are listed more briefly in comparison to the "package-lock.json" file. It includes the name and version numbers of the used dependencies. Listing 5.5 is a sample "package.json" file.

```
{
  "name": "sample",
  "version": "6.5.0",
  "description": "A sample description",
  "license": "MIT",
  "dependencies": {
    "got": "^9.6.0",
    "registry-auth-token": "^4.0.0",
    "registry-url": "^5.0.0",
    "semver": "^6.2.0"
  }
}
```

Listing 5.5: A sample package.json file

The names and the version numbers are passed to the "Vulnerability-Checker" in order to find matching CPE entries.

The "JavaScriptScanner" on the other hand, receives all ".js" files included in the project except those that belong to the libraries downloaded by NPM. NPM libraries are by default located in the "node_modules" folder and its sub folders. In order to distinguish libraries from normal source code, the first block comment is processed for a version number. See 4.3.2.The regex format used for this purpose is `''v\d+\.\d+\.\d+``.For ".js" libraries, the file name is used as the library name. The library and version number of ".js" files is also passed over to the "VulnerabilityChecker".

With both NPM and ".js" library info available, the process of matching library information to CPE entries starts. The structure of a CPE entry is as follows:

```
cpe:/{part}:{vendor}:{product}:{version}:{update}:{edition}:{
    language}
```

Listing 5.6: The structure of the Common Platform Enumeration (CPE)

The ”part“ entry is set to ”a“ for applications. The options ”h“ and ”o“ are references to hardware and operating system vulnerabilities respectively. An example for a CPE entry for the ”GNU C Library“, commonly known as ”glibc“ is the following:

```
cpe:/a:gnu:glibc:2.28
```

Listing 5.7: CPE entry regarding glibc version 2.28

The CPE entries will be matched to the library data and the results are handed over to the ”AnalysisManager“ and saved to the JIRA database.

5.3 Relevant Database Schema in JIRA

In JIRA plugin development, it is good practice to use the internal JIRA database instead of an external one. The interface used to access the JIRA database is called ”Active Objects“. According to the official Atlassian SDK Development website, ”The goal of the Active Objects plugin is to provide a plugin data storage component that plugins can and should use to persist their private data.“ Active objects are used in ProDynamics to store and retrieve the results of the vulnerability detection.

Figure 5.1 shows the database schema of the JIRA tables which are used behind the scenes to save the results of vulnerability detection. The ”SECURITY_AO“ table keeps track of projects and their associated sprints. The table ”SECURITY_FILE_AO“, lists all files belonging to a project. The table ”SECURITY_VC_AO“ contains the vulnerabilities found in the libraries of the projects. The CVE id and its corresponding information is saved in this table. The ”SECURITY_SCCD_AO “ table is used to save the information regarding vulnerability in source code files. It contains data not only regarding the CVE Id and its corresponding information but also the line number and location where the match has been detected in the source code and the security code repository. All the above data will be available to the graphical interface which reports back to the user, the vulnerabilities detected in a project.

5.4 JSON Support for the NVD Database

The national vulnerability database has announced that the XML vulnerability feed will retire on October 9th, 2019 and cease to exist on this

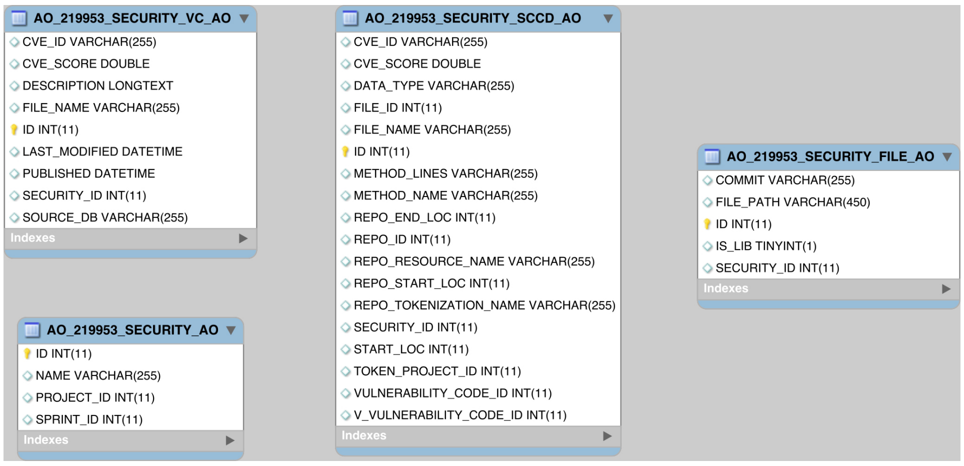


Figure 5.1: Schema of the relevant part of the JIRA database

data. Therefore it was crucial to update the NVD database extraction of the ProDynamics plugin to support the JSON feed. For this purpose, the ”CVEParser“ class was rewritten with an object based rather than the original contextual based parsing of the XML feed. The library ”Gson“ by Google is used in this project for JSON parsing. The ”CVEJSONReader“ class is called in the initialization phase to download the NVD database. This class calls the ”CVEJsonParser“ to parse the JSON feed of the NVD database into corresponding objects. An online tool called ”jsonschema2pojo“ was used to generate the necessary JSON objects for the parser [18]. The ”NVDUpdater“ class saves the parsed vulnerability data to the corresponding SQLite database which is later used in the library checking and code clone detection. Figure 5.2 shows the schema of the local SQLite database. The schema of the local database is modeled after the structure of the original XML feed.

5.5 Functional and Graphical Improvements

The original implementation of the ”Security Checker“ plugin is slightly revised in terms of functionality and graphical user interface. The settings menu of the ProDynamics plugin contained a text field which was used to point to the specific sub-folder where the vulnerability code repository was located. With the implementation of JavaScript support, the code repository was divided in Java and JavaScript sub-folders for the programming languages respectively. Additionally, with unexpected folder structure changes in the repository, the settings data had to be adjusted in all instances of ProDynamics used on different machines. Hence, in order to support multiple programming languages and to tackle the

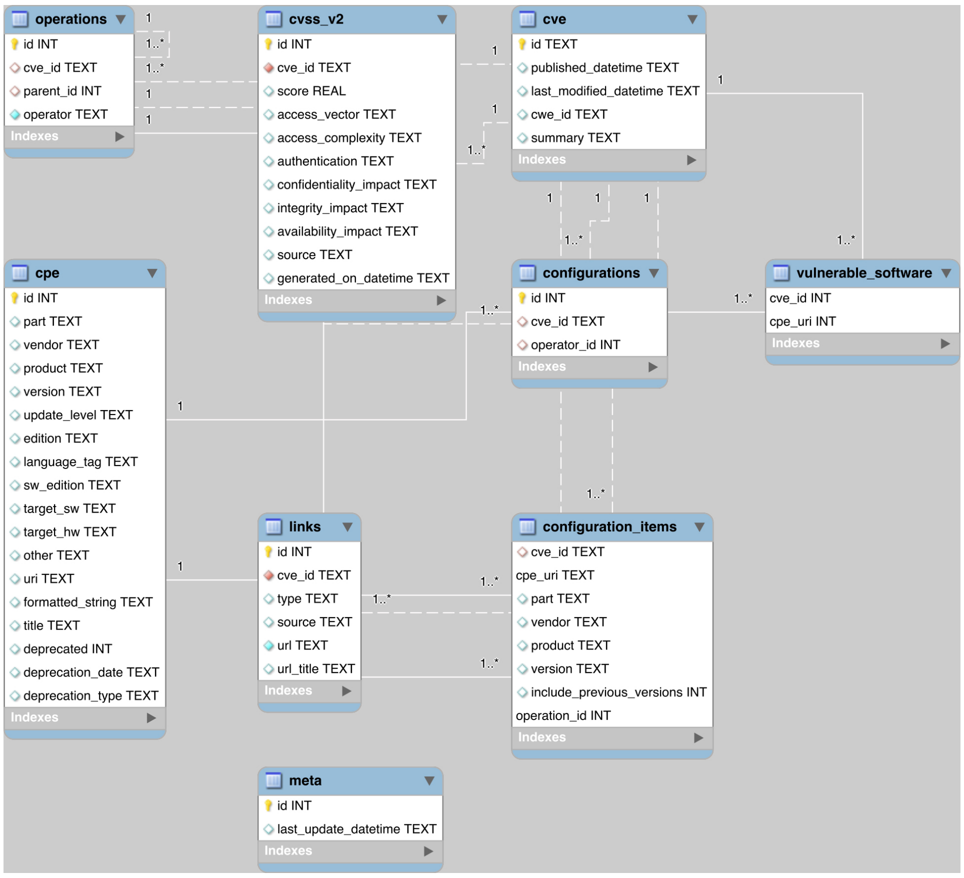


Figure 5.2: Schema of the local SQLite NVD database

above mentioned problem with the folder structure, the path to different programming languages will now be parsed from a file inside the root of the chosen repository. This file which has to specifically be named ”language_paths.json“, needs to be in the JSON file format. It will include the paths to the repositories of the supported programming languages. Listing 5.8 shows the default syntax of the ”language_paths.json“ file.

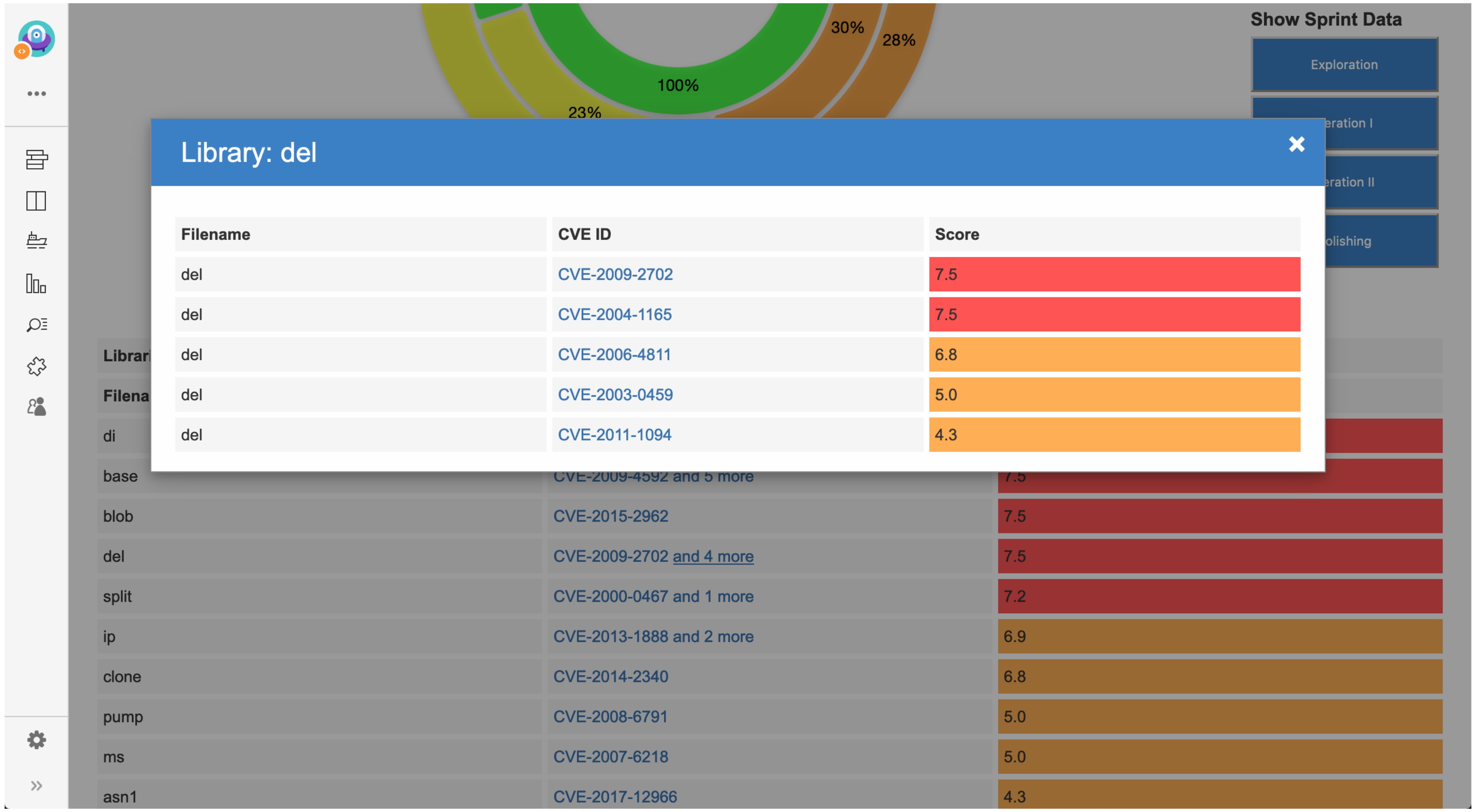
```json
{
  "path_to_languages":{
    "Java":"Java/Mixed/results",
    "JavaScript":"JavaScript/Version1GitHub/results"
  }
}
```

Listing 5.8: A sample language_paths.json file

In order to report back vulnerability data to the user, the found library and source code vulnerabilities are listed in separate tables. The table entries are modified to be arranged by the most severe security risk based on the

CVE score of the vulnerabilities. For each library and source code file the most significant security risk is shown and by clicking on the corresponding link, other vulnerabilities regarding the selected file is presented to the user. This results in a more compact and comprehensive structure of the GUI and creates a neater user experience.

In the original version of the "Security Checker", the created tables only included the vulnerabilities which were found in the last sprint. Since an overview of the vulnerabilities found in the previous sprints might be handy to some developers, buttons were added to the GUI which generate the vulnerability tables according to the data committed in the course of that specific sprint. Figure 5.3 is a sample for the redesigned structure of the GUI.



Filename	CVE ID	Score
del	CVE-2009-2702	7.5
del	CVE-2004-1165	7.5
del	CVE-2006-4811	6.8
del	CVE-2003-0459	5.0
del	CVE-2011-1094	4.3

Librar		
Filena		
di		
base	CVE-2009-4592 and 5 more	7.5
blob	CVE-2015-2962	7.5
del	CVE-2009-2702 and 4 more	7.5
split	CVE-2000-0467 and 1 more	7.2
ip	CVE-2013-1888 and 2 more	6.9
clone	CVE-2014-2340	6.8
pump	CVE-2008-6791	5.0
ms	CVE-2007-6218	5.0
asn1	CVE-2017-12966	4.3

Figure 5.3: Redesigned GUI of the ”Security Checker“

5.6 Unexpected Difficulties

The scripting language JavaScript is constantly updated. See 2.4. This update to the syntax and semantic, created a challenge for the implementation of the tokenizer. Most of the parsers written in Java for the programming language JavaScript like the well known "Rhino" by Mozilla, only support JavaScript up to ECMAScript 5 (Version 5) and some features of "ECMAScript 2015" (Version 6) [13]. The library "Closure-Compiler" by Google, has a built in JavaScript parser. Unfortunately, it does not have a proper documentation for its built in parsing functionality. Therefore it took some time until I was able to create a working JavaScript parser written purely in Java.

The other significant hurdle I faced, was the development of the JSON parser for the NVD feed. The structure of the soon to be deprecated XML feed, was not identical to that of the JSON feed. Although the underlying information was for the most part identical, the structure was changed slightly. This meant that some of the information which was received from the JSON feed had to be processed before it could be passed on to the already available architecture of the ProDynamics plugin. As an example, all possible CPE entries for each application was directly included in the XML feed. In the JSON feed, the products which had vulnerabilities in more than one version, used a "*" in the corresponding CPE entry and referenced the version number separately. Therefore, although the JSON feed is more efficiently structured, the initialization process now processes the entries and adds the CPE data accordingly. Another hardship in this regard was the syntax of the CPE entry. The JSON feed uses CPE version 2.3. Since the "Security Checker" plugin is based on CPE version 2.2, the entries had to be reprocessed to version 2.2.

The JIRA plugin development package which is released and maintained by Atlassian has a few flaws that make the process of development unnecessary time consuming. The Atlassian SDK which is required for JIRA plugin development, offers "QuickReload" as a tool to assist developers to modify and run cycles faster. Unfortunately, uncaught exceptions which are not rare in Java software development sometimes lead to total shut down of JIRA as a whole. Since the "Security Checker" makes use of threads to run vulnerability checking in parallel to the normal use of the JIRA software, an uncaught exception often led to a total halt to the development process before restarting the server anew. It also has to be noted that on each run, the whole JIRA software has to be initialized completely.

Chapter 6

Evaluation

The ”Security Checker“, as part of the JIRA plugin ”ProDynamics“, aimed at detecting vulnerabilities in the programming language Java. The main goal of this thesis is to extend the ”Security Checker“ to also support JavaScript through a corresponding JavaScript code clone detector and library checker. In order to evaluate the implemented software, known vulnerabilities are imported and analyzed by the software and the results together with the corresponding performance indicators is described in this chapter.

6.1 Performance Indicators

Precision, recall and F1 are some of the standard indicators for the performance of an information retrieval (IR) system [8]. Table 6.1 can be used as a basis in order to understand these indicators and their corresponding equations.

	Relevant	**Nonrelevant**
Retrieved	True Positives (TP)	False Positives (FP)
Not Retrieved	False Negatives (FN)	True Negatives (TN)

Table 6.1: Contingency table for performance indicators [19]

According to Manning et al. ”Recall (R) is the fraction of relevant documents that are retrieved“ [19]. The corresponding equation for calculating recall is:

$$Recall(R) = \frac{TP}{TP + FN}$$

Manning et al. also point out ”Precision (P) is the fraction of retrieved documents that are relevant“ [19]. The equation for calculating precision in an information retrieval system is:

$$Precision(P) = \frac{TP}{TP + FP}$$

Information retrieval systems may have their focus on different performance indications. A high recall might be more important while searching for files on the hard drive of a computer. On the other hand, a high recall might lead to a huge number of results which might make the search uninterpretable. A high precision can also cause similar difficulties. Therefore, a combined performance indicator where recall and precision have a weighted effect on the calculated value, can be a better indicator for the performance of a specific information retrieval system. Manning et. al. define the F measure as such: "A single measure that trades off precision versus recall is the F measure, which is the weighted harmonic mean of precision and recall" [19]. The equation for the F measure is as follows:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad \text{where} \quad \beta^2 = \frac{1-\alpha}{\alpha} \quad \text{and} \quad \alpha \in [0,1]$$

The popular F_1 measure aka. "balanced F measure" weighs recall and precision equally thus $\alpha = \frac{1}{2}$ and $\beta = 1$. In course of this chapter, the evaluation of the code clone detector and library checker is based on the "balanced F measure" or F_1.

6.2 Evaluating Code Clone Detection

Apart from the effectiveness of the underlying technique used by SourcererCC, code clone detection for JavaScript depends on token extraction and the way code blocks are defined. Therefore it is reasonable to evaluate the implementation of JavaScript code clone detection separately.

To evaluate code clone detection in JavaScript, 20 vulnerable code snippets, as listed in Table 6.2, are used. These sample code snippets are added to the security repository. For each sample, the code sample itself, the appropriate fix, a manually prepared Type 2 and a manually prepared Type 3 code clone is processed with the developed software. The original "Security Checker" plugin offers three different settings for code clone detection:

- High Recall
- Combined Precision & Recall
- High Precision

As part of the evaluation, all three configurations have been examined. The results of the code clone detection will be presented in course of this chapter.

	CVE	Score	Product
1	CVE-2019-11358	4.3	jQuery
2	CVE-2017-2445	4.3	Apple Safari
3	CVE-2017-1000042	6.1	Mapbox Project
4	CVE-2014-7192	10.0	syntax-error Node.js Module
5	CVE-2014-7192	10.0	syntax-error Node.js Module
6	CVE-2018-3754	6.5	query-mysql Node.js Module
7	CVE-2018-16462	8.4	apex-publish-static-files Node.js Module
8	CVE-2016-3714	8.4	ImageMagick
9	CVE-2018-3750	7.5	deep_extend Node.js Module
10	CVE-2007-3670	4.3	Microsoft Internet Explorer
11	CVE-2017-16088	10.0	safe-eval Node.js Module
12	CVE-2017-11895	7.5	Microsoft Internet Explorer
13	CVE-2014-0046	2.6	Ember.js
14	CVE-2018-3774	10.0	url-parse Node.js Module
15	CVE-2018-11093	6.1	CKEditor 5
16	CVE-2017-7534	5.4	OpenShift Enterprise
17	CVE-2019-7167	7.5	Zcash
18	CVE-2015-1840	5.0	jquery_ujs.js in jquery-rails
19	CVE-2015-7384	7.5	Node.js
20	CVE-2019-10744	7.5	loadash Node.js Module

Table 6.2: 20 Vulnerable Source Code Samples. Multiple code snippets refer to the same CVE ID

6.2.1 Essential Code Samples

An example for a vulnerable code snippet is listing 6.1. It is the corresponding code snippet to CVE-2014-7192. It is also a code clone of type-1 to itself. This vulnerability can lead to the "eval" function being misused by remote attackers to execute arbitrary code on the victims computer. A vulnerability with a base score of 10.0 creates a high risk for a software artifact. According to the national vulnerability database, the original description of this vulnerability is as follow:

”Eval injection vulnerability in index.js in the syntax-error package before 1.1.1 for Node.js 0.10.x, as used in IBM Rational Application Developer and other products, allows remote attackers to execute arbitrary code via a crafted file.“

```
var http = require('http');
http.createServer(function (request, response) {
  if (request.method === 'POST') {
    var data = '';
    request.addListener('data', function(chunk) {
        data += chunk;
    });
    request.addListener('end', function() {
      var bankData = eval("(" + data + ")");
      bankQuery(bankData.balance);
   });
  }
});
```

Listing 6.1: Vulnerable code snippet according to CVE-2014-7192

The ”eval“ function if not handled correctly can be used to define variables in the scope it is used in. Therefore, it might become a target for exploitation. In order to fix this vulnerability, the JavaScript function ”eval“ has to be run in ”strict mode“ (Introduced in ECMAScript 5). In this case, the ”eval“ function can also be replaced by ”JSON.parse“. Listing 6.2 is a fix for the mentioned vulnerable code snippet.

```
var http = require('http');
http.createServer(function (request, response) {
  "use strict"
  if (request.method === 'POST') {
    var data = '';
    request.addListener('data', function(chunk) {
        data += chunk;
    });
    request.addListener('end', function() {
      var bankData = JSON.parse(data);
      bankQuery(bankData.balance);
   });
  }
});
```

Listing 6.2: Corresponding fix for listing 6.1

In software development, code with similar output can be written in variations of Syntaxes. A code clone detector needs to be able to not only detect exact matches but also similar code snippets. Listing 6.3 is a type-2 code clone of the above vulnerable code snippet. In the type-2 clone for listing 6.1, variables ”http“ and ”bankData“ have been renamed to ”link“and ”bankD“ respectively. A line comment has also been added to line number

6.

```
1  var link = require('http');
2  link.createServer(function (request, response) {
3    if (request.method === 'POST') {
4      var data = '';
5      request.addListener('data', function(chunk) {
6          data += chunk;
7      });
8      //a sample comment
9      request.addListener('end', function() {
10       var bankD = eval("(" + data + ")");
11       bankQuery(bankD.balance);
12    });
13   }
14 });
```

Listing 6.3: Type-2 clone for listing 6.1

Listing 6.4 is a type-3 clone for the above example. Here, the original variable ”link“ has been omitted and replaced by a direct call to the require function ”require(’http’)“. Similar to the type-2 clone, the variable ”bankData“ has also been renamed to ”bankD“. A comment on line 4 also differentiates this code snippet from the original listing 6.1.

```
1  require('http').createServer(function (request, response) {
2    if (request.method === 'POST') {
3      var data = '';
4      //a sample comment
5      request.addListener('data', function(chunk) {
6          data += chunk;
7      });
8      request.addListener('end', function() {
9        var bankD = eval("(" + data + ")");
10       bankQuery(bankD.balance);
11    });
12   }
13 });
```

Listing 6.4: Type-3 clone for listing 6.1

6.2.2 Code Clone Detection Analysis

In the conducted evaluation, fixes that are categorized as code clones are constantly viewed as non relevant but retrieved (false positive) information. On the other hand, the aim of the code clone detector is to identify code clones of types 1 2 and 3 and therefore categorizing such detection as relevant and retrieved (true positive) data. It is also worth mentioning that recall is calculated based on the known information about the vulnerability of each code snippet. If it were calculated without any information regarding the code samples, the calculation of the recall value would be more complex [2]. Table 6.3 is an overview of the code clone detection with regard to each fix and code clones of different type.

It is worth mentioning that the aim of code clone detection is to find code that may produce a similar vulnerability. Code snippets that contain the fixed version of a known vulnerability should not be designated as vulnerable source code. The found code clones are designated in the following table with a ✔ sign which is a symbol for a true positive (TP) and fixes that are detected as code clones are identified with ✖ as their detection is considered a false positive (FP). Blank columns indicate code samples which have not been found. In case of fixes, the blank columns are true negatives (TN) and in case of the different code types, they refer to false negatives (FN).

	High Recall				Combined Precision & Recall				High Precision			
	Fix	Type-1	Type-2	Type-3	Fix	Type-1	Type-2	Type-3	Fix	Type-1	Type-2	Type-3
1	✖	✔	✔	✔	✖	✔			✖	✔		
2	✖	✔	✔	✔	✖	✔	✔	✔	✖	✔		
3	✖	✔	✔	✔	✖	✔			✖	✔		
4	✖	✔	✔	✔	✖	✔	✔		✖	✔	✔	
5	✖	✔	✔	✔	✖	✔	✔		✖	✔		
6	✖	✔	✔	✔	✖	✔	✔		✖	✔		
7	✖	✔	✔	✔	✖	✔	✔			✔	✔	
8	✖	✔	✔	✔	✖	✔	✔		✖	✔		
9	✖	✔	✔		✖	✔	✔		✖	✔		
10	✖	✔	✔	✔	✖	✔	✔		✖	✔		
11	✖	✔	✔	✔	✖	✔	✔		✖	✔		
12	✖	✔	✔	✔	✖	✔	✔		✖	✔		
13	✖	✔	✔	✔	✖	✔	✔		✖	✔		
14	✖	✔	✔	✔	✖	✔				✔		
15	✖	✔	✔	✔	✖	✔	✔			✔		
16	✖	✔	✔	✔	✖	✔	✔		✖	✔		
17	✖	✔	✔	✔	✖	✔	✔		✖	✔		
18	✖	✔	✔		✖	✔	✔		✖	✔		
19	✖	✔	✔		✖	✔	✔		✖	✔		
20	✖	✔	✔	✔	✖	✔				✔		

Table 6.3: Code clone detection results. Row number correspond to the vulnerabilities listed in table 6.2.

In contrast to the evaluation result of code clone detection using SourcererCC for the programming language Java which was conducted by F. P. Viertel et al. [28], the different code block structure for JavaScript results in many fixes being identified as code clones. Table 6.4 lists the values for recall, precision and a combination of both for all the code clone detection settings available to the "Security Checker". For better comprehension and in order to be able to compare results, fixes which were identified as code clones are also designated as true positives. This has to be kept in mind that with regard to table 6.4, the goal of a clone detector has to be, maximizing the performance indicators of all code clone types while minimizing the performance indicators of fixes. For the calculation of the indicators, true

		High Recall	Combined Precision & Recall	High Precision
Type-1 Clones	TP	20	20	20
	R	100	100	100
	P	100	100	100
	F_1	100	100	100
Type-2 Clones	TP	20	16	2
	R	100	80	10
	P	100	100	100
	F_1	100	88	18
Type-3 Clones	TP	17	1	0
	R	85	5	0
	P	100	100	100
	F_1	91	9	0
Fixes	TP	20	20	16
	R	100	100	80
	P	100	100	100
	F_1	100	100	88

Table 6.4: Recall, precision and F_1 according to table 6.3

positives are defined as code samples that are identified as code clones. The reason why precision is calculated as 100% is that for each analysis iteration, the input code snippets are all part of the expected code clone result. Therefore it is evident that precision is calculated as 100%.

6.3 Evaluating Library Detection

In the original implementation of the "Security Checker" plugin, library vulnerability detection for the programming language Java, depends on three different pieces of information which is extracted from the manifest file of Java libraries identified through their file extension, "jar". This information are library name, vendor and version. The programming language JavaScript lacks such a manifest file. Thus, information has to be extracted from the comment blocks or through a package manager like NPM. This process is discussed with more detail in section 4.2.

In order to evaluate library vulnerability detection using information gathered through NPM, a mix of known vulnerable libraries mixed with a list of libraries which have not been identified as vulnerable, are used. The list of libraries which have been used for the evaluation purpose consists of 35 libraries. The vulnerable libraries consist of 15 outdated NPM packages. Whilst the updated version of these libraries may have been fixed, older versions with known vulnerabilities were purposefully chosen. For libraries with no known vulnerability, the 20 most depended upon NPM packages are used [23]. Table 6.5 shows an overview of the 15 vulnerable NPM libraries which were used for the evaluation.

	CVE	**Score**	**Product**	**Version**
1	CVE-2018-3724	5.0	general-file-server	1.1.8
2	CVE-2018-16461	9.8	libnmap	0.4.10
3	CVE-2016-10551	9.8	waterline-sequel	0.5.0
4	CVE-2018-1999024	5.4	mathjax	2.7.3
5	CVE-2018-3713	6.5	angular-http-server	1.9.0
6	CVE-2016-10548	6.1	reduce-css-calc	1.2.4
7	CVE-2015-1164	4.3	serve-static	0.1.2
8	CVE-2015-8862	6.1	mustache	2.2.0
9	CVE-2016-10539	7.5	negotiator	0.6.0
10	CVE-2017-10910	6.5	mqtt	0.3.13
11	CVE-2018-0114	7.5	node-jose	0.9.2
12	CVE-2017-1000220	9.8	pidusage	1.1.4
13	CVE-2017-16023	7.5	decamelize	1.1.0
14	CVE-2017-5858	5.9	converse.js	2.0.4
15	CVE-2017-16021	6.5	uri-js	2.1.1

Table 6.5: 15 vulnerable NPM libraries

The above vulnerable libraries are mixed with the latest version of the 20 most depended upon NPM packages. None of these libraries have, as of this writing, been identified as vulnerable by the NVD database. Table 6.6 is the list of the 20 most depended upon NPM libraries with stand August 2019 which have been mixed with the above table for the evaluation purpose.

	Product	**Version**		**Product**	**Version**
1	lodash	4.17.15	11	async	3.1.0
2	chalk	2.4.2	12	fs-extra	8.1.0
3	request	2.88.0	13	bluebird	3.5.5
4	react	16.9.0	14	tslib	1.10.0
5	express	4.17.1	15	axios	0.19.0
6	commander	2.8.1	16	uuid	3.3.3
7	moment	2.24.0	17	underscore	1.9.1
8	debug	4.1.1	18	vue	2.6.10
9	prop-types	15.7.2	19	classnames	2.2.6
10	react-dom	16.9.0	20	mkdirp	0.5.1

Table 6.6: 20 most depended upon NPM libraries

For the evaluation, if at least one CVE Id is identified by the library checker for a given library, it is considered as vulnerable. The corresponding recall, precision and F_1 values of the conducted searches for all three configuration is listed in table 6.7.

	High Recall	Combined Precision & Recall	High Precision
TP	15	15	12
FP	4	4	0
FN	0	0	3
R	100	100	80
P	79	79	100
F_1	88	88	89

Table 6.7: Recall, precision and F_1 for library detection

The package manager NPM has set a few constraints for developers who wish to upload their libraries to the package manager. One of these constraints is the name of the package. NPM does not allow duplicate package names. Therefore each developer has to come up with a unique name for his/her library [24]. This has led to the library checker to score a high precision and F_1 value for settings event set to "High Recall".

Chapter 7

Summary and Outlook

Throughout this chapter, the basic idea of this thesis and the results developed based on it are presented. It will also contain an outlook and some relevant ideas for the further development of the "Security Checker" plugin.

7.1 Summary

In course of this work, the ”Security Checker“ of the ProDynamics plugin, which was developed to detect vulnerabilities in Java projects, was extended to also support JavaScript. A purely Java based, JavaScript tokenizer was developed to enhance code clone detection for JavaScript. In order to detect source code vulnerabilities in JavaScript, the method-based clone recognition was revised based on the specific requirements of JavaScript. This includes analyzing source code not only based on functions but also classes and code blocks outside of functions. Furthermore, library vulnerability checking was implemented for JavaScript with regard to the popular package manager NPM. Since JavaScript source code and libraries both use the ”.js“ extension, JavaScript files are examined for a corresponding version number in the comment blocks and processed as libraries accordingly.

With code blocks defined a bit differently in JavaScript in comparison to Java, ”SourcererCC“ is still able to detect code clones of up to type-3 when the emphasize in code clone detection is on high recall values. While moving towards a higher precision, type-3 code clones start to become unrecognizable. This shows that because of the different code block definition in JavaScript in comparison to Java, source code is more sensitive to variations.

Duplicated naming is not allowed in the package manager NPM. Therefore library names are added to the library checker as whole. This has led to high precision and F_1 values for all possible configurations of the ”Security Checker“.

The GUI of the "Library Checker" was also improved to enhance the user experience and to offer the user a more comprehensive picture of possible vulnerabilities. Since the XML interface of the NVD vulnerability database will soon be suspended, the JSON feed was adapted to ensure future compatibility with the fundamental requirements of the "Security Checker".

7.2 Outlook

Although "SourcererCC" offers a solid basis for code clone detection, lexical analysis does still have the downside of generating a high number of false positives. In order to tackle this problem, condition based vulnerability detection as explained in the paper by W. Jimenez et al. might be a proper extension to the current approach [11]. Since static techniques for vulnerability detection in source code is principally based on a databases of known vulnerable code, an open and free to use database similar or in extension to the NVD database can be of help to many developers. With a community of experts accessing and updating this database, it can play a significant role in future projects including the "Security Checker" of ProDynamics.

Bibliography

[1] Atlassian. The world's best teams work better together with atlassian. `https://www.atlassian.com/customers`.

[2] W. Brunotte. Security Code Clone Detection entwickelt als Eclipse Plugin. Masterarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2018.

[3] N. C. Corporation. Netscape and sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet. `https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html`. 28 industry-leading companies to endorse JavaScript as a complement to Java for easy online application development.

[4] R. Data. Javascript versions. `https://www.w3schools.com/js/js_versions.asp`. W3Schools.

[5] GitHub. Most starred javascript repositories. `https://github.com/search?l=JavaScript&q=stars%3A%3E40000&type=Repositories`.

[6] GitHub. Trending. `https://github.com/trending`.

[7] Google. What is the closure compiler? `https://developers.google.com/closure/compiler/`.

[8] C. Goutte and E. Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In D. E. Losada and J. M. Fernández-Luna, editors, *Advances in Information Retrieval*, pages 345–359, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] S. G. Hubert Garavel and A.-M. Leventi-Peetz. Formal methods for safe and secure computers systems. Technical Report BSI Study 875, Federal Office for Information Security, 2013.

[10] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[11] W. Jimenez, A. Mammar, and A. Cavalli. Software vulnerabilities, prevention and detection methods: A review 1. 07 2010.

[12] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: A web vulnerability scanner. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 247–256, New York, NY, USA, 2006. ACM.

[13] W. Kapke. Rhino es2015 support. `http://mozilla.github.io/rhino/compat/engines.html`.

[14] F. Kortum, J. Klünder, and K. Schneider. Behavior-driven dynamics in agile development: The effect of fast feedback on teams. 05 2019.

[15] I. V. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University West Lafayette, IN, USA, West Lafayette, IN, USA, 1998. AAI9900214.

[16] D. Lee. Whatsapp discovers 'targeted' surveillance attack. `https://www.bbc.com/news/technology-48262681`. CVE ID: CVE-2019-3568.

[17] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Softw.*, 11(5):23–30, Sept. 1994.

[18] J. Littlejohn. Generate plain old java objects from json or json-schema. `http://www.jsonschema2pojo.org`.

[19] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[20] M. Matthaei. Integration und test von sicherheitsüberprüfungen in java. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2019.

[21] MITRE. Terminology. `https://cve.mitre.org/about/terminology.html`.

[22] Mozilla. Functions. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions`.

[23] npm. most depended upon packages. `https://www.npmjs.com/browse/depended`.

[24] npm. solving npm's hard problem: naming packages. `https://blog.npmjs.org/post/116936804365/solving-npms-hard-problem-naming-packages`.

[25] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[26] K. Schwaber and J. Sutherland. The scrum guide. `https://www.scrumguides.org/scrum-guide.html`.

[27] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: An empirical study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 61–72, May 2016.

[28] F. P. Viertel, W. Brunotte, D. Strüber, and K. Schneider. Detecting security vulnerabilities using clone detection and community knowledge. In *Proceedings of the Thirty-First International Conference on Software Engineering and Knowledge Engineering (SEKE'19)*. KSI Research Inc., 2019.

[29] F. P. Viertel, F. Kortum, L. Wagner, and K. Schneider. Are third-party libraries secure? a software library checker for java. In A. Zemmari, M. Mosbah, N. Cuppens-Boulahia, and F. Cuppens, editors, *Risks and Security of Internet and Systems*, pages 18–34, Cham, 2019. Springer International Publishing.

[30] T. Watanabe, M. Akiyama, F. Kanei, E. Shioji, Y. Takata, B. Sun, Y. Ishi, T. Shibahara, T. Yagi, and T. Mori. Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 14–24, Piscataway, NJ, USA, 2017. IEEE Press.

[31] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, Santa Clara, CA, Aug. 2019. USENIX Association.

Listings

List of Tables

List of Figures