Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Integration und Test von Sicherheitsüberprüfungen in JIRA

Integration and Test of Security Monitoring in JIRA

Bachelorarbeit

im Studiengang Informatik

von

Martin Matthaei

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Joel Greenyer
Betreuer: M. Sc. Fabien Patrick Viertel

Hannover, 07.04.2019

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 07.04.2019

Martin Matthaei

Zusammenfassung

Computersysteme durchdringen den Alltag, wobei die eingesetzte Software dabei immer komplexer wird und Abhängigkeiten zu Drittanbieterkomponenten steigen. Sicherheitslücken können weitreichende Folgen haben, weshalb diese frühestmöglich erkannt und beseitigt werden sollten. Außerdem ist es wichtig Entwickler für Schwachstellen im eigenem Programmcode sowie eingesetzten Fremdkomponenten zu sensibilisieren. Werkzeuge zum automatischen Erkennen von Schwachstellen sollten deshalb bereits im Entwicklungsprozess integriert werden.

Im Rahmen dieser Bachelorarbeit wurde ein Plugin für JIRA entwickelt, das den Quellcode und die verwendeten Bibliotheken von Java-Projekten auf bekannte Schwachstellen untersucht. Die Analyse findet innerhalb der JIRA-gestützten, agilen Entwicklungsabläufe statt. Schwachstellen werden auf Basis der Daten National Vulnerability Database nach ihrer Schwere klassifiziert und dem Nutzer grafisch dargestellt. Dies bietet Entwicklern die Möglichkeit gefundene Probleme einzuschätzen und zu verstehen. Durch die Integration können die Ergebnisse der Analyse so direkt in die Entwicklungsabläufe, die JIRA bereitstellt, einfließen.

Abstract

Integration and Test of Security Monitoring in JIRA

Computer systems pervade everyday life, whereas used software gets more complex and dependencies to third party components are increasing. Software vulnerabilities may have major consequences. Therefore, they should be detected and elimenated as early as possible. Moreover, it is important to raise developers' awareness on vulnerabilities in their own source code as well as used foreign components. Tools for automatic vulnerability detection should therefore be integrated into the development process.

This bachelor thesis describes the development of a JIRA plugin that analyses source code and libraries of Java projects for known vulnerabilities. The analysis happens within the agile development processes supported by JIRA. Results are classified and visually presented upon their severity based on National Vulnerability Database data. Thus developers have the possibility to evaluate and apprehend detected issues. Through the JIRA integration, analysis results can directly incorporated into the next development cycle.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Problemstellung . . . 1
- 1.2 Lösungsansatz . . . 2
- 1.3 Struktur der Arbeit . . . 2

2 Grundlagen **3**
- 2.1 Schwachstelle . . . 3
- 2.2 National Vulnerability Database . . . 4
- 2.3 Code Clones . . . 4
- 2.4 Versionskontrollsysteme . . . 6
- 2.5 JIRA . . . 7

3 Konzept **9**
- 3.1 Komponenten . . . 9
  - 3.1.1 Vulnerability Checker . . . 9
  - 3.1.2 Security Code Clone Detector . . . 10
  - 3.1.3 Security Code Repository . . . 10
  - 3.1.4 ProDynamics Plugin . . . 11
- 3.2 Integration der Komponenten . . . 13
  - 3.2.1 Analyse . . . 13
  - 3.2.2 Speicherung der Ergebnisse . . . 14
  - 3.2.3 Projektdaten . . . 14
  - 3.2.4 Lokales Dateisystem . . . 15

4 Softwaretests **16**
- 4.1 Funktionale Tests . . . 16
- 4.2 Durchführung . . . 16
- 4.3 Ergebnis . . . 18

5 Implementierung **19**
- 5.1 Architektur . . . 19
- 5.2 AnalysisManager . . . 20
- 5.3 Git-Integration . . . 21
- 5.4 REST-API . . . 22

5.5 Frontend und Benutzeroberfläche 23
5.5.1 Analyseergebnis 23
5.5.2 Administration 24
5.5.3 Konfiguration 25
5.6 Unvorhergesehene Schwierigkeiten 27

6 Evaluation **29**
6.1 Durchführung 29
6.2 Auswertung 30
6.2.1 Java-Quellcode 32
6.2.2 Java-Bibliotheken 35

7 Verwandte Arbeiten **39**

8 Zusammenfassung und Ausblick **41**
8.1 Zusammenfassung 41
8.2 Ausblick 42

Kapitel 1

Einleitung

Computer durchdringen unseren Alltag. Selbst in Dingen, in denen wir es mitunter nicht vermuten wie Fahrkartenautomaten oder Kaffeemaschinen, steckt ein Mikroprozessor, der Programmcode ausführt, welcher wiederum Fehler enthalten kann.

Die Auswirkungen durch Sicherheitslücken, die sich dadurch ergeben, sind vielfältig. Während Lücken in Banking-Apps einen finanziellen Schaden hervorrufen, können Softwarefehler in Medizintechnik Menschenleben kosten. Im Zuge der sogenannten „Industrie 4.0“ ist es möglich, dass Sicherheitslücken in Industriesteuerungen zu Sabotagezwecken missbraucht werden. Laut Branchenverband *Bitkom* entsteht der deutschen Wirtschaft so ein jährlicher Schaden in Höhe von 55 Milliarden Euro[7].

Die eingesetzte Software wird dabei immer komplexer und besitzt mitunter Abhängigkeiten zu Softwarebibliotheken, die wiederum selbst Schwachstellen besitzen können. Dabei stellt der Anteil verwendeter quelloffener Komponenten in kommerzieller Software nicht selten über 75% des Quellcodes dar[14].

Allgemein ist es deshalb wichtig, Sicherheitslücken in Software frühestmöglich festzustellen und zu beheben, so Pham et al. [10]. Dabei müssen nicht nur eigener Programmcode, sondern auch externe Abhängigkeiten berücksichtigt werden.

1.1 Problemstellung

Das manuelle Finden sicherheitskritischer Fehler ist keine triviale Aufgabe und fordert einiges an Expertise in diesem Bereich [6]. Erst der regelmäßige Umgang mit Sicherheitstools und -konzepten schafft die nötige Erfahrung und somit auch ein Bewusstsein für sicherheitsrelevante Programmierung [19].

Zusätzlich dazu muss durch Softwarebibliotheken eingebundener Programmcode aus externen Quellen ebenfalls auditiert werden. Die Biblio-

theken selbst können wiederum Abhängigkeiten zu anderen Bibliotheken besitzen. Mit jeder neuen Version können neue Schwachstellen im Quellcode entstanden sein. Dementsprechend wären Bibliotheken mit jeder Aktualisierung einer erneuten Überprüfung zu unterziehen. Teilweise liegt deren Quellcode allerdings gar nicht offen und die eigene Prüfung ist somit nicht realisierbar.

Darüber hinaus erfordert die manuelle Überprüfung laut Brunotte einen hohen zeitlichen Aufwand [6] und außerdem eine Integration in bestehende Prozessabläufe. Der damit verbundene logistische und finanzielle Aufwand wird oft gescheut, da er keinen unmittelbar spürbaren Mehrwert bringt.

1.2 Lösungsansatz

Es wird ein Plugin zur automatischen Suche von Sicherheitslücken für die Projektmanagement-Software JIRA entwickelt. Dieses sucht und visualisiert auf Basis bekannter Sicherheitslücken mögliche kritische Fehler in Projekten der Programmiersprache Java. Grundlage hierfür bilden der *Security Code Clone Detector* von Wasja Brunotte [6] und der *Library Checker* von Leif Erik Wagner [18].

Das Plugin gibt bereits während des Entwicklungsprozesses Hinweise über potentiell sicherheitskritischen Programmcode sowie eingebundene Bibliotheken. Weiter wird der Entwickler dabei unterstützt die Schwere gefundener Probleme einzuschätzen. Hierzu werden Informationen zum Ursprung der gefundenen Fehler und die Bewertung der Kritikalität mittels zugewiesener CVE-Wertung (s. Abschnitt 2.2) angegeben.

Durch das direkte Feedback wird ein Bewusstsein für verwundbaren Quellcode geschaffen. Somit wird ein Lernprozess beim Entwickler angestoßen, um zukünftige Fehler im Voraus zu vermeiden.

1.3 Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert. In Kapitel 1 wird auf die Problemstellung und den Lösungsansatz dieser Arbeit eingegangen. Des Weiteren wird ihre Struktur beschrieben. Kapitel 2 stellt die Grundlagen zum Verständnis der Arbeit dar.

Das Konzept für das JIRA-Plugin wird in Kapitel 3 erläutert. Dabei wird auf die zu integrierenden Komponenten eingegangen. Das Kapitel 4 beschreibt auf die durchgeführten Software-Tests im Bezug auf *Security Code Clone Detector* und *Vulnerability Checker*. Im Kapitel 5 wird dann die Implementierung des Plugins beschrieben. Darauf folgt Kapitel 6 mit einer Evaluationsstudie zu Schwachstellen in studentischen Softwareprojekten mit anschließendem Überblick auf verwandte Arbeiten in Kapitel 7. Abschließend folgen Zusammenfassung und Ausblick in Kapitel 8.

Kapitel 2

Grundlagen

In diesem Kapitel werden Grundlagen vermittelt, die das weitere Verständnis dieser Arbeit nötig sind. Dazu werden die Begriffe Schwachstelle, CVE und Code Clone erläutert. Danach wird auf die Programme Git und JIRA eingegangen.

2.1 Schwachstelle

Für de Schwachstelle bzw. englisch Vulnerability gibt es in der Informationssicherheit verschiedene Definitionen. Die Norm *ISO/IEC 27005:2008* zum Thema *Information security risk management* definiert den Begriff Vulnerability wie folgt:

> „A weakness of an asset or group of assets that can be exploited by one or more threats.“ [17]

Das National Institute of Standards and Technology (NIST) fasst seine Definition nicht ganz so breit:

> „A flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy.“ [11]

Im allgemeinen kann man zusammenfassen, dass eine Schwachstelle ein Softwareproblem darstellt, durch das eine Schadwirkung auf ein System ausgeübt werden kann. Schwachstellen liegen meist einem Programmierfehler zugrunde, können jedoch auch absichtlich vom Entwickler beispielsweise in Form einer Hintertür platziert werden. Durch die Definition des NIST wird deutlich, dass nicht jeder ausnutzbare Fehler eine Schwachstelle darstellt, sondern erst dann als solche gilt, sobald dadurch die Sicherheit des Systems beeinflusst wird.

2.2 National Vulnerability Database

Die National Vulnerability Database (NVD)[1] ist eine vom US-amerikanischen Staat durch die Behörde NIST bereitgestellte Datenbank zum Abruf von öffentlichen Sicherheitslücken in Software. Diese werden durch weitere Informationen ergänzt. Dazu gehören die verwundbaren Programmversionen sowie Ratschlägen und Lösungen zur Behebung oder Verhinderung ihrer Ausnutzung. Die Datenbasis bilden dabei die Common Vulnerabilities and Exposures (CVE).

CVE ist ein Standard zur einheitlichen Einstufung öffentlich bekannter Softwareschwachstellen. Für eine eindeutige Kennzeichnung werden CVE-IDs, bestehend aus CVE-Prefix, Jahreszahl sowie einer fortlaufenden Nummer (z. B. CVE-2011-0766), vergeben.

Darüber hinaus gibt die NVD eine Bewertung der Schwere einer Sicherheitslücke mittels Common Vulnerability Scoring System (CVSS) an. Unter Berücksichtigung von u.a. Komplexität der Ausnutzbarkeit und Auswirkungen wird dieser Schweregrad errechnet. Der mögliche Wertebereich reicht dabei von 0.0 bis 10.0, wobei 10.0 für die schwerst mögliche Schwachstelle steht. Die Schwachstellen werden ihrer CVSS-v2.0-Wertung entsprechend, wie in Tabelle 2.1 dargestellt, in die Kategorien „low“, „medium“ und „high“ eingruppiert [2].

CVSS v2.0 Ratings	
Severity	Base Score Range
Low	0.0 – 3.9
Medium	4.0 – 6.9
High	7.0 – 10.0

Tabelle 2.1: CVSS v2.0 *Qualitative Severity Rating Scale*

Zusätzlich gibt es die neuere Bewertung nach CVSS v3.0, die sich anders berechnet und die Kategorie „critical“ für den Wertebereich 9.0 bis 10.0 einführt. Da Scoring-Werte nach CVSS v3.0 nur für Schwachstellen ab 2015 verfügbar sind, wird, um die Vergleichbarkeit der Werte für ältere Schwachstellen zu gewährleisten, in dieser Arbeit nur mit Scores nach v2.0 gearbeitet.

2.3 Code Clones

Der Begriff Code Clones beschreibt Fragmente von Programmcode, die einander ähneln. Die Ähnlichkeit kann sowohl auf semantische wie auch syntaktische Weise bestehen.

[1] National Vulnerability Database: `https://nvd.nist.gov/`

Dabei werden nach Sheneamer et al. [16] folgende Typen unterschieden (für eine anschauliche Darstellung siehe Abb. 2.1):

Type-1 (Exact clones) Bis auf den Whitespace und eventuelle Kommentare sind die Fragmente identisch.

Type-2 (Renamed/Parameterized) Zwei Code-Fragment sind abgesehen der Namen von Variablen, Typen, Literalen und Funktionen ähnlich.

Type-3 (Near miss clones/Gapped clones) Die Fragmente sind ähnlich, besitzen zusätzlich zu Type-2 aber Änderungen wie hinzugefügte oder entfernte Statements.

Type-4 (Semantic clones) Code-Fragmente, die sich semantisch ähnlich sind ohne syntaktische Ähnlichkeit zu besitzen.

Ursprüngliches Code Fragment CF_0

```
for(i = 0; i < 10; i++)
{
    // foo 2
    if (i % 2 == 0)
        a = b + i;
    else
        // foo 1
        a = b - i;
}
```

CF_1 – Typ-1 Clone

```
for(i = 0; i < 10; i++)
{
    if (i % 2 == 0)
        a = b + i; // comment 1
    else
        b = b - i; // comment 2
}
```

CF_2 – Typ-2 Clone

```
for(j = 0; j < 10; j++)
{
    if (j % 2 == 0)
        y = x + j; // comment 1
    else
        y = x - j; // comment 2
}
```

CF_3 – Typ-3 Clone

```
for(i = 0; i < 10; i++)
{
    // new statement
    a = 10 * b;
    if (i % 2 == 0)
        a = b + i; // comment 1
    else
        a = b - i; // comment 2
}
```

CF_4 – Typ-4 Clone

```
while(i < 10)
{
    // a comment
    a = (i % 2 == 0) ? b + i : b - i;
    i++;
}
```

Abbildung 2.1: Code Clones nach Typ, Brunotte [6]

Code Clones können durch das Kopieren und wiederverwenden von Programmcode entstehen. Sie können sicherheitsrelevante Konsequenzen haben, wenn die Quelle bereits Fehler besitzt. Das Aufspüren von Code Clones der verschieden Typen ist dabei nicht trivial, da keine syntaktische Ähnlichkeit bestehen muss.

2.4 Versionskontrollsysteme

Ein Versionskontrollsystem (VCS) ist ein System zu Verwaltung von Änderungen. In der Softwaretechnik bezieht sich dies meist auf Modifikationen in Quellcode oder zugehöriger Dokumentation bzw. Konfiguration. Durch die Protokollierung von Veränderungen, ergibt sich einerseits die Möglichkeit der Attribution jener, andererseits die Wiederherstellung beliebiger Entwicklungsstände. Des Weiteren wird durch ein VCS der gemeinsame Zugriff auf Daten koordiniert.

In der Softwareentwicklung stellen Subversion (SVN) und Git die meistgenutzten Systeme zur Versionskontrolle dar [5]. Es handelt sich bei beiden Programmen um freie Software. SVN wurde 2000 von CollabNet als Nachfolger für die Versionskontrollsoftware CVS entwickelt. Bei Git handelt es sich dagegen um eine Neuentwicklung. Sie wurde ursprünglich von Linus Torvalds 2005 zum Quellcode-Management für die Entwicklung des Linux-Kernels erschaffen, besitzt heute aber darüber hinaus durch Source-Code-Hoster wie GitHub[2] große Popularität bei Open-Source-Entwicklern. Der Hauptunterschied zwischen diesen ist, dass es sich bei Git um ein verteiltes VCS, bei SVN dagegen um ein zentrale Versionsverwaltung handelt.

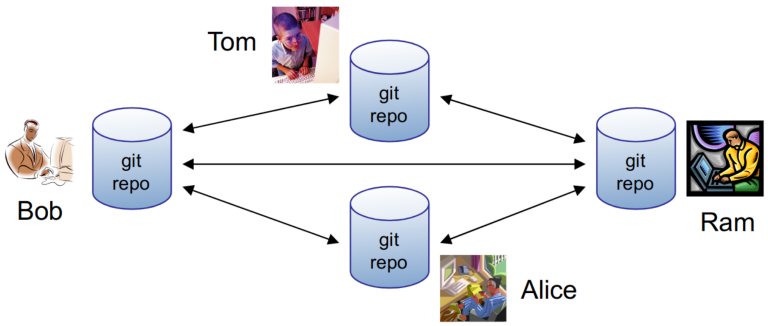

Abbildung 2.2: Dezentrale Nutzung von Git, Bird et al. [5]

Zentrale Versionsverwaltung setzt auf Client-Server-Prinzip. Es existiert ein einziges zentrales Repositorium (engl. Repository), auf das alle Nutzer gemeinsam zugreifen. Die Versionsgeschichte befindet sich dabei ausschließlich in besagtem Repository.

[2] `https://github.com/`

Bei der verteilten Versionsverwaltung gibt es hingegen keinen zentralen Server. Jeder Benutzer besitzt eine lokale Kopie mit vollständiger Historie, in der er seine Änderungen vornimmt. Diese Änderungen können wiederum in ein beliebiges anderes Repositorium eingespielt werden. Diese Arbeitsweise wird in Abb. 2.2 an Git verdeutlicht. Jedes lokale Repository ist gleichberechtigt. Sie können miteinander verglichen und Änderungen können untereinander eingespielt werden. Aus organisatorischen Gründen kann es jedoch sinnvoll sein auch hier ein Repository zu erstellen, das wie ein zentrales als Referenz für die lokalen Repositorien dient. So wird die Synchronisation unter den Nutzern vereinfacht, da ausschließlich die Änderungen dieses Repositoriums verfolgt werden müssten.

2.5 JIRA

JIRA ist eine proprietäre Web-basierte Anwendung der Firma *Atlassian* zur Organisation und Verwaltung von Softwareprojekten. Die Software hilft dabei Aufgaben zu kategorisieren, priorisieren und entsprechend zu delegieren und verfügt darüber hinaus über Issue- und Bug-Tracking-Funktionen. Die Interaktion findet über den Webbrowser des Nutzers statt.

Das Prozessmanagement ist dabei vor allem auf agile Softwareentwicklungsmodelle wie Scrum ausgerichtet. Laut offiziellem *The Scrum Guide*stellt Scrum einen empirischen Prozess mit einem iterativen, inkrementiellen Ansatz dar, die Erfahrungen als Grundlage nimmt, um Vorhersehbarkeit zu optimieren und Risiken zu kontrollieren [3]. Die Entwicklung findet dabei in zeitlichen Intervallen namens Sprints statt. Für einen Sprint steht ein festgelegtes Budget bereit, das den maximalen Arbeitsaufwand festlegt. Zu Beginn eines Sprints wird der Aufwand der zu erledigenden Aufgaben für den definierten Zeitabschnitt abgeschätzt. Diese werden dem kommenden Sprint zugeordnet bis das festgelegte Budget ausgereizt ist. Die restlichen Aufgaben verbleiben im sogenannten Backlog und werden in einem späteren Sprint abgearbeitet. Während des Sprints werden keine weiteren Aufgaben hinzugefügt, sondern die vorhandenen abgearbeitet. Am Ende des Sprints wird dieser ausgewertet und dementsprechend der Folgesprint geplant.

JIRA ist auf dieses Entwicklungsmodell zugeschnitten. Es besitzt Werkzeuge zum Planen von Sprints mit Möglichkeiten zum Festlegen und Nachverfolgen des jeweiligen Arbeitsaufwands. Die zugehörigen Aufgaben werden in Form vom Tickets verwaltet und an die Nutzer verteilt. Darüber hinaus bietet JIRA diverse Möglichkeiten zur Auswertung wie Diagramme zum Sprintverlauf, um Arbeitsabläufe zu optimieren.

Es handelt bei JIRA um eine Java-basierte Webanwendung in Form eines Java Web Archives, die auf Webservern, die Unterstützung für Java-Servlets bieten, lauffähig ist. Die Anwendung nutzt diverse quelloffene Komponenten und Frameworks, die miteinander agieren und die JIRA-eigenen Funktionen

im Kern der Anwendung bereitstellen. Eine ausführliche Übersicht dazu bietet die *JIRA Server Developer Documentation*[3].

Durch den modularen Aufbau lässt sich JIRA mit Plugins um zusätzliche Funktionalitäten wie beispielsweise die Integration von Versionskontrollsystemen (s. vorherigen Abschnitt 2.4) erweitern. Diese lassen sich über den Atlassian Marketplace[4] beziehen. Darüber hinaus bietet JIRA eine ausführliche REST-API[5], über die sich umfangreiche Daten abfragen lassen.

[3] JIRA Architecture overview: `https://developer.atlassian.com/server/jira/platform/architecture-overview/`

[4] Atlassian Marketplace: `https://marketplace.atlassian.com/`

[5] JIRA REST-API: `https://developer.atlassian.com/cloud/jira/software/rest/`

Kapitel 3

Konzept

Es wird ein JIRA-Plugin entwickelt, in das Security Code Clone Detector und Vulnerability Checker integriert werden. Dazu werden in Abschnitt 3.1 dieses Kapitels die zur Integration nötigen Komponenten erläutert. Danach wird in Abschnitt 3.2 darauf eingegangen, wie genau das Zusammenspiel der einzelnen Komponenten untereinander sowie den durch JIRA bereitgestellten Schnittstellen erfolgt.

3.1 Komponenten

3.1.1 Vulnerability Checker

Der Vulnerability Checker stellt ein Eclipse-Plugin als Ergebins der Masterarbeit *Konzept und Entwicklung eines Schwachstellenprüfers für Java-Bibliotheken* von Leif Erik Wagner [18] dar. Dieses vergleicht in ein Eclipse-Projekt eingebundene Java-Bibliotheken mit bekannten Schwachstellen in der National Vulnerability Database. Durch die Einbindung in Eclipse wird die Prüfung auf veraltete und unsichere Bibliotheken in den Entwicklungsprozess integriert.

Die Zuordnung bekannter Schwachstellen findet dabei über die Metadaten des JAR-Manifestes der Bibliothek und ihren Dateinamen statt. Hierfür werden die verfügbaren Informationsfelder zu Hersteller, Produkt und Version des Manifests genutzt. Der Abgleich dieser Daten wird mit einer lokalen Kopie der NVD durchgeführt, die bei jedem Programmstart automatisch über den offiziellen XML-Feed aktualisiert wird.

Der Vulnerability Checker überprüft allerdings ausschließlich Bibliotheken, die sich als JAR-Datei im Ordner eines Java-Projektes befinden. Bei der Arbeit mit Build-Management-Systemen wie Maven oder Gradle werden externe Abhängigkeiten erst beim Programmbau aufgelöst und heruntergeladen. In einem solchen Fall ist die Prüfung durch den Vulnerability Checker nicht möglich.

3.1.2 Security Code Clone Detector

Um Entwicklern während der Programmierung von Java-Anwendungen automatisiert Rückmeldung bezüglich möglicher Schwachstellen zu geben, hat Wasja Brunotte im Rahmen der Masterarbeit *Security Code Clone Detection entwickelt als Eclipse Plugin* den Security Code Clone Detector entwickelt [6].

Die Grundlage für das Auffinden von Schwachstellen bilden Code Clones, wobei die Typen 1 bis 3 erkannt werden. Code-Fragmente, die bekannte Schwachstellen aufweisen, dienen dabei als Datenbasis für die Clone-Detection. Diese sind im *Security Code Repository* (s. folgenden Abschnitt 3.1.3) gesammelt und mit Metadaten versehen für die Analyse aufbereitet. Beim Start des Plugins wird der Quellcode der Schwachstellen in Form von Tokens als Referenz hinterlegt. Erstellt der Nutzer Java-Code, wird dieser ebenfalls tokenisiert und verglichen. Dieser Vorgang läuft methodenbasiert ab, d. h. der Quellcode-Vergleich wird auf der Ebene von Java-Methoden durchgeführt. Die so gefundenen Code Clones werden dann dem Nutzer mitsamt Informationen zur entsprechenden Schwachstelle angezeigt. Diese basieren auf den Metadaten, die das Security Code Repository bereitstellt. Ist dort eine CVE-ID hinterlegt, werden zusätzliche NVD-Daten aus der lokalen Kopie angezeigt. Dieser Ablauf ist in Abb. 3.1 aus jener Masterarbeit schematisch dargestellt.

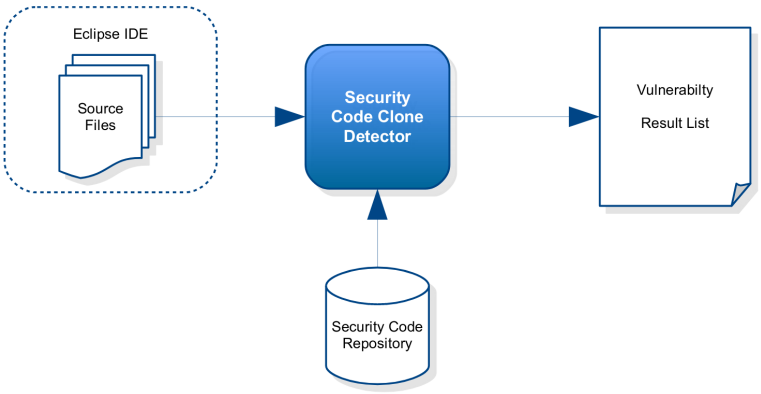

Abbildung 3.1: Schematische Darstellung Security Code Clone Detector, Brunotte [6]

3.1.3 Security Code Repository

Das Security Code Repository basiert auf den Ergebnissen der Bachelorarbeit *Security Code Exporter für Github* von Christian Müller [13]. Der Exporter durchsucht der Source-Code-Hoster Github[1] nach sicherheitsrelevantem

[1] Github: `https://github.com/`

Quellcode. Über Verweise auf CVE-IDs werden Änderungen bekannten Schwachstellen zugeordnet. Die resultierende Metadaten zu Verwundbarkeit und Verbesserung werden in einer Datenbank, dem Security Code Repository, gespeichert und dienen den Security Code Clone Detector als Grundlage zum Aufspüren von sicherheitskritischen Clones.

Zusätzlich wurde das Repository um die Ergebnisse der Bachelorarbeit *Suche von sicherheitsrelevantem Code auf Stack Overflow* von Yannick Evers [9] erweitert. Dieser nutzt die Frageplattform Stack Exchange[2] als Datenbasis. Laut Evers werden in den zugehörigen Diskussionen allerdings sehr selten CVEs erwähnt. In diesem Fall kann somit keine automatische Klassifizierung über NVD-Daten erfolgen.

Das Repository ist wie folgt aufgebaut: Es enthält Java-Dateien mit bekannten Schwachstellen im Unterordner `VULNERABILITY`. Das Verzeichnis `FIX` enthält die zugehörigen Verbesserungen mit gleichem Dateinamen wie die Schwachstelle. Ist ein Exploit bekannt, findet sich dieses im gleichnamigen Verzeichnis. Die entsprechenden Metadaten liegen in einer SQLite-Datenbank, deren Schema in Abb. 3.2 dargestellt wird.

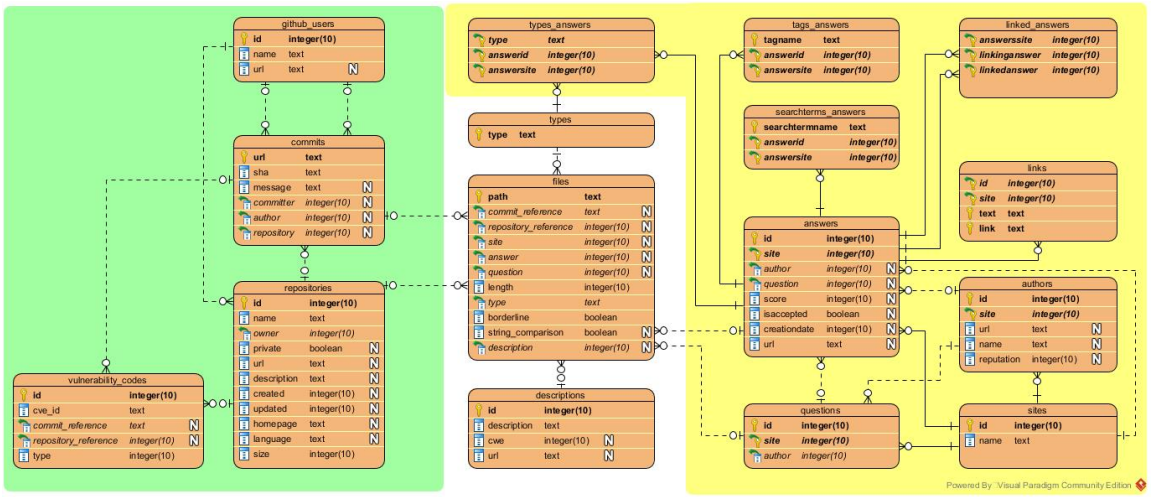

Abbildung 3.2: Datenbankschema des Security Code Repositorys basierend auf den Arbeiten von Evers [9] und Müller [13]

3.1.4 ProDynamics Plugin

Beim ProDynamics Plugin handelt sich um eine Erweiterung für die Projektverwaltungssofware JIRA. Es analysiert verschiedene Dynamiken eines Projekts und stellt diese grafisch dar. Darüber hinaus ist eine Fragebogen-Funktion integriert, über die Teammitglieder unter anderem zur Stimmung

[2] Stack Exchange: `https://stackexchange.com/`

innerhalb des Projekts befragt werden. Diese Daten verknüpft mit dem Daten, die JIRA bereitstellt, werden für verschieden Analysen genutzt. Dazu werden Korrelationen und Abhängigkeiten zwischen Projektdaten aus JIRA und Faktoren wie Stimmung der Entwickler berechnet. Durch diesen objektiven, datenbasierten Ansatz, können Dynamiken und Tendenzen gefunden werden, die Menschen sonst verborgen bleiben, so Klünder et al. [12].

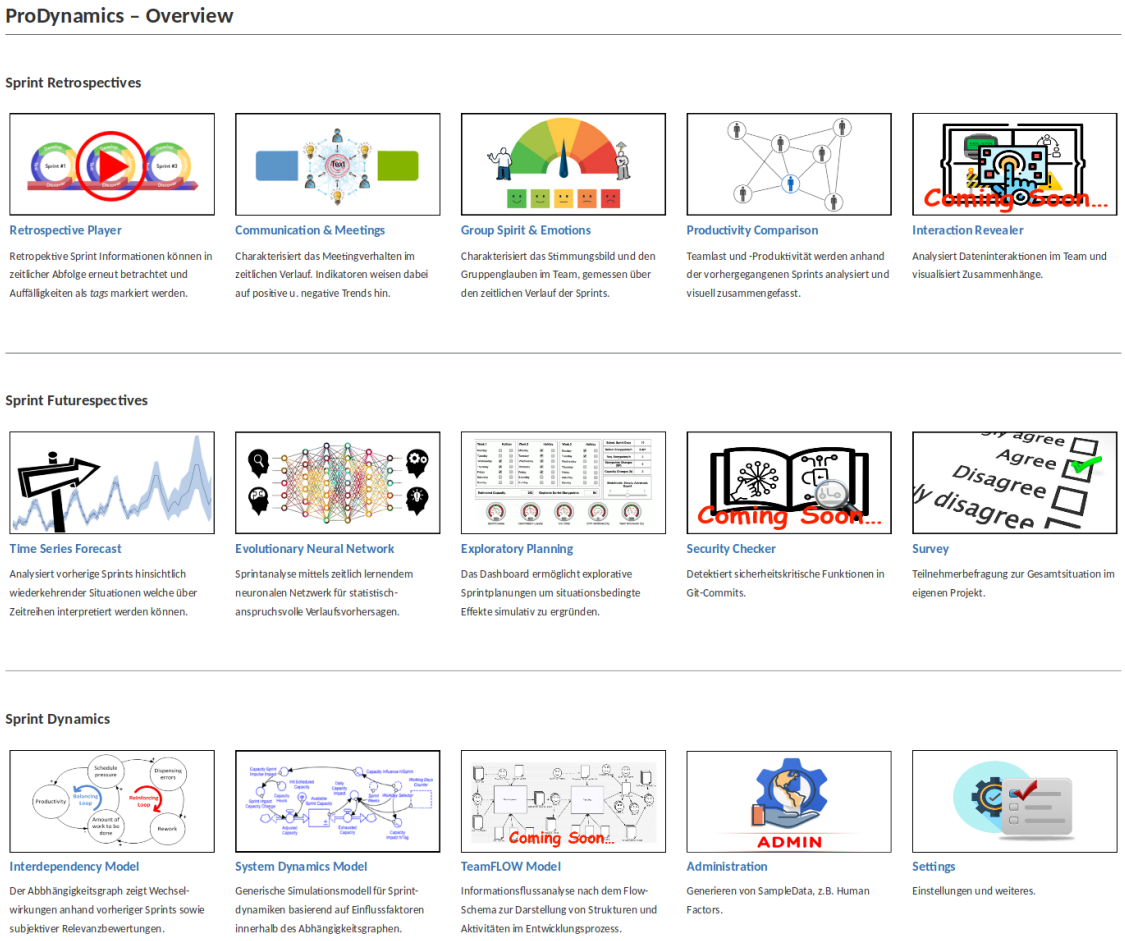

Abbildung 3.3: Übersicht ProDynamics-Plugin

Die Funktionen sind, wie Abb. 3.3 zeigt, in drei Bereiche unterteilt: Sprint Retrospectives, Sprint Futurespectives und Sprint Dynamics.

Die Retrospektive enthält Werkzeuge um Prozesse innerhalb der bereits abgeschlossenen Sprints zu visualisieren. Beispiele stellen die Darstellungen des Stimmungsbildes im Team im Verlauf eines Sprints oder der Vergleich der Produktivität der Teammitglieder untereinander dar.

Unter Futurspektive sind entsprechenden Funktionen zur Vorhersage und Planung kommender Sprints angesiedelt. Hierzu gehört auch die

Teilnehmerbefragung, die Teammitglieder regelmäßig ausfüllen sollen. Komplexe Abhängigkeiten und Wechselwirkungen werden in Form von Graphen dargestellt und sind in der Kategorie Sprint Dynamics zu finden.

Das in dieser Arbeit entwickelte Security Plugin wird nicht als eigenständiges Plugin umgesetzt, sondern als Komponente innerhalb des ProDynamics Plugins integriert. Somit werden die Analysemöglichkeiten der Projektdynamik um sicherheitsrelevante Funktionen erweitert.

3.2 Integration der Komponenten

Das Konzept sieht vor, Security Code Clone Detector und Vulnerability Checker, die zuvor als Plugin für die Eclipse-IDE entwickelt wurden, in Form eines JIRA-Plugins umzusetzen. Dazu werden alle vorhanden Abhängingkeiten zu Eclipse entfernt. Dies betrifft vor allem die Benutzeroberfläche und Konfiguration, deren Funktionalität für JIRA re-implementiert bzw. teilweise neu entwickelt werden müssen.

Da die Plugins das MVC-Pattern verwenden, besteht die Grundidee darin, Model- und Controllerklassen beizubehalten und die entsprechenden Views durch Webinterfaces auf HTML- und JavaScript-Basis zu ersetzen. Damit das über das Webfrontend die Controllerklassen angestreuert werden können, wird eine REST-API implementiert.

Aufgrund des modularen Aufbaus von Security Code Clone Detector und Vulnerability Checker können Views und andere Abhängigkeiten zu Eclipse-SDK-Bibliotheken weitestgehend unkompliziert entfernt werden. Die Plugins haben jeweils eine Controllerklasse, die Eclipse als Einstiegspunkt dient und alle benötigten Methoden bereitstellt. Beim Vulnerability Checker ist dies die Klasse `VulnerabilityChecker`, beim Security Code Clone Detector die Klasse `CloneDetectorManager`. Diese Klassen werden beibehalten und als Schnittstelle für die Analyse genutzt. Die weitere interne Struktur der Programme wird bei der Implementierung nicht berücksichtigt und als Blackbox behandelt.

Bei der Integration muss zudem darauf geachtet werden, dass mögliche Konflikte zwischen eingesetzten Bibliotheken behoben werden. Da JIRA selbst einen hohe Zahl von Abhängigkeiten mitbringt, können Konflikte diesbezüglich nicht ausgeschlossen werden. Security Code Clone Detector und Vulnerability Checker benutzen jeweils den SQLite-JDBC-Driver für ihre interne Datenbankverbindung. Für eine konfliktfreie Verwendung in JIRA muss dieser explizit geladen werden.

3.2.1 Analyse

Es wird ein Controller entwickelt, die Analyse verwaltet. Dieser nimmt Projektdaten entgegen und bereitet diese für die Analyse mit Security Code

Clone Detector und Vulnerability Checker entsprechend vor. Weiter übergibt er die Daten an die oben genannten Klassen und speichert das Ergebnis.

Vor der Analyse müssen die Datenbanken, die die Grundlage für die Schwachstellensuche bilden, synchronisiert werden. Der Vulnerability Checker bezieht seine Daten aus der NVD. Der Security Code Clone Detector nutzt die Code-Fragmente im Security Code Repository für die Clone-Suche und verknüpft deren Metadaten ebenfalls mit Schwachstellendaten aus der NVD. Während sich der Vulnerability Checker bei Initialisierung selbst um Aktualisierung seines Datensatzes kümmert, müssen den Code Clone Detector die Daten bereitgestellt werden. Somit muss vor Analysestart das Security Code Repository in das lokale Dateisystem kopiert und der Vulnerability Checker initialisiert werden, damit der Security Code Clone Detector dessen NVD-Datenbank mitnutzen kann.

3.2.2 Speicherung der Ergebnisse

JIRA abstrahiert Datenbankinteraktionen über *Active Objects*, der Zugriff erfolgt über *Active Object Handler*. Die Ergebnisse der Analyse werden getrennt für Vulnerability Checker und Security Code Clone Detector gespeichert. Außerdem wird ein Active Object mit den Projektdaten angelegt, dessen Schlüssel von den Analysedaten referenziert wird. Somit können Projekteigenschaften und Analyseergebnisse einander zugeordnet werden. Darüber hinaus wird eine Liste geänderter Dateien gespeichert. Diese dient bei der Auswertung zur Feststellung der Anzahl an Dateien ohne Befunde, da in den Analyseergebnissen nur Dateien mit Schwachstellen enthalten sind.

3.2.3 Projektdaten

Die Analyse des Quellcodes soll den iterativen Entwicklungszyklus entsprechend am Ende jedes Sprints stattfinden, sodass die Befunde in die Planung des Folgesprints eingehen können. Die Datenbasis für die dazu nötigen Informationen, wie die Zuordnung von Projekten und Sprints sowie die zugehörigen Zeiträume, liefert JIRA. Über dessen REST-API[3] lassen sich sämtliche Projektinformationen abfragen.

In JIRA selbst findet jedoch von Haus aus nur die Organisation und Verwaltung der Projekte statt. Eine Verknüpfung mit Versionskontrollsystemen zur Verwaltung des Quellcodes ist nicht integriert. Dazu wird das Plugin *Git Integration for Jira* von BigBrassBand[4] verwendet. Durch die somit vorhandene Verknüpfung zwischen Projektdaten und Versionskontrolldaten, kann eine sprintspezifische Analyse des Quellcodes erfolgen. Start- und Enddatum des betreffenden Sprints definieren welche Änderungen in der Analyse zu berücksichtigen sind. Diese werden mithilfe des VCS in den

[3] JIRA REST-API: `https://developer.atlassian.com/cloud/jira/software/rest/`
[4] BigBrassBand *Git Integration for Jira*: `https://bigbrassband.com/`

Projektquellen ermittelt und der entsprechende Versionsstand im Dateisystem wiederhergestellt, um darauf die Analyse mit Security Code Clone Detector und Vulnerability Checker zu starten. Somit werden jeweils nur geänderte Dateien analysiert, wodurch sich wiederum der Zeitaufwand für die Schwachstellensuche verringert.

3.2.4 Lokales Dateisystem

Für die Durchführung der Security-Analyse müssen Daten im lokalen Dateisystem abgelegt werden. Dies schließt folgende Daten ein: die lokale Kopie des Security Code Repository, die NVD-Datenbank, lokale Kopien von Repositorien der analysierten Projekte sowie Datenbanken und temporäre Dateien von Security Code Clone Detector und Vulnerability Checker. Diese Dateien speichert das Plugin im Ordner `data/security-plugin` des JIRA-Home-Verzeichnisses. Den Pfad dort hin stellt JIRA über die Methode `getHome()` der Java-Klasse `JiraHome` bereit.

Kapitel 4

Softwaretests

Vor der Integration von Security Code Clone Detector und Vulnerability Checker, werden deren Funktion durch Softwaretests ausführlich geprüft. Die verwendeten Methoden werden in diesem Kapitel erläutert.

4.1 Funktionale Tests

Die Funktionalität eines Systems wird über seine Spezifikation und funktionalen Anforderungen definiert. Funktionale Tests stellen die Prüfung eines Systems auf diese Anforderungen dar. Hierbei wird das Sollverhalten wie das Gegenverhalten für die Erfüllung der funktionalen Voraussetzungen geprüft.

Im Gegensatz dazu stehen nicht-funktionale Tests, wie Performanztests, Stabilitätstests oder Usabilitytests. In dieser Arbeit werden im Rahmen der Integration von Security Code Clone Detector und Vulnerability Checker ausschließlich die Prüfung der funktionalen Anforderungen durchgeführt. Zur Durchführung der Softwaretests wird Java-Framework JUnit[1] verwendet.

4.2 Durchführung

Da es sich bei den vorliegenden Programm bereits um fertig implementierte Software handelt, ist der ausführliche Test jeder einzelnen Komponente mit sehr hohem Aufwand verbunden, da die ursprünglichen Designentscheidungen hinter internen Abläufen im komplexen Zusammenspiel verschiedener Komponenten nicht mehr nachvollziehbar sind, um diese sinnvoll zu testen. Die somit entstandenen Abhängigkeiten verhindern den isolierten Test einzelner Module. Deshalb werden die Programme hier jeweils als Gesamtsystem ohne Rücksicht auf ihre interne Struktur in Blackbox-Tests überprüft.

Der Fokus liegt beim Testen auf den Schnittstellen, die von JIRA-Plugin genutzt werden sollen. Beim Security Code Clone Detector ist dies die Klasse

[1] JUnit: `https://junit.org`

`CloneDetectorManager` im Paket `de.luh.se.sccd.plugin.conedetector`, der über die Methode `searchClones` Tokenisierung und Clone-Suche abstrahiert. Der Vulnerability Checker stellt seine Funktionen zur Analyse von Bibliotheken über die Methoden des Controllers `VulnerabilityChecker` bereit. Die Klasse ist dabei so konzeptioniert, dass Abläufe aufeinander aufbauen und von einander abhängig sind, sodass der Test einzelner Methoden nicht ohne weiteres möglich ist. Hier bieten sich ein Systemtest an, bei dem der Prozessablauf im Produktiveinsatz simuliert wird. Hierfür wird ein Set von Testdaten erzeugt mit dem die Tests durchgeführt werden. Listing 1 zeigt beispielhaft den vereinfachten Ablaufen eines durchgeführten Tests am Vulnerability Checker: erst wird der Datenstruktur eine Bibliothek hinzugefügt, anschließend nach Schwachstellen gesucht und überprüft, ob eine erwartete Schwachstelle in der Liste der Ergebnisse enthalten ist. Zum Schluss wird die gefundene Schwachstelle auf die Blacklist gesetzt und überprüft, dass sie nicht mehr unter den gefundenen Schwachstellen auftaucht. Die einzelnen Schritte sind dabei voneinander abhängig. Die Blacklist-Methode ist so implementiert, dass die Schwachstelle als Objekt benötigt wird, das zuvor bei der Schwachstellensuche erzeugt worden ist.

```
1  @Test
2  public void testLibrary() {
3    String lib = "guava-23.0.jar"
4    VulnerabilityChecker.addLib(lib);
5    VulnerabilityChecker.getSearchJob();
6
7    String cve = "CVE-2018-10237";
8    VulnerabilityModel vulnerabilities =
     ↪  VulnerabilityChecker.getVulnerabilityModel();
9    assertNotNull(vulnerabilities);
10   assertTrue(vulnerabilities.contains(cve));
11
12   VulnerabilityChecker.blacklistVulnerability(vulnerabilities.get(cve));
13   assertFalse(VulnerabilityChecker.getVulnerabilityModel().contains(cve));
14 }
```

Listing 1: Vereinfachtes Beispiel Systemtest am Vulnerability Checker

Der Security Code Clone Detector besteht aus eigenständigen Komponenten wie Tokenizer und Sourcerer, auf denen die Clone Detection aufbaut. Diese Komponenten werden jeweils über ihre Controller getestet. Anschließend findet der Test bezüglich der Integration über den `CloneDetectorManager` statt. Wie beim Vulnerability Checker bauen die internen Abläufe aufeinander auf, sodass bestimmte Schritte im Analyseablauf einander voraussetzten.

Wo die Möglichkeit besteht und Grenzwerte für die Eingabe bekannt sind, werden entsprechende Grenzwerttests durchgeführt. Beispielsweise ist der kleinstmöglich Wert für von der NVD bereitgestellte Listen nach Jahreszahl das Jahr 2002. Somit definiert der Wert 2002 die untere und das aktuelle Jahr die obere Grenze. Mit diesem Wissen kann das Verhalten des Programms auf Eingaben außerhalb dieser Grenzen getestet werden.

4.3 Ergebnis

Insgesamt wurden durch das Testen keine gravierenden funktionalen Fehler gefunden. Die gefundenen Fehler wurden im Rahmen der Integration der beiden Komponenten in das JIRA-Plugin entsprechend behoben. Teilweise reagieren die Programme sensibel, falls Ordner, in den Konfigurations- oder temporäre Daten abgelegt werden sollen, nicht existieren. Dies wird abgefangen, indem die Existenz zuvor geprüft und bei Nichtvorhandensein die entsprechenden Verzeichnisse erstellt werden.

Kapitel 5

Implementierung

In diesem Kapitel wird die Umsetzung des vorangegangenen Konzeptes beschrieben. Hierzu wird zunächst auf die Architektur und Schnittstellen eingegangen. Es folgen Ausführungen zu grafischer Oberfläche und Details bezüglich der Konfiguration. Zuletzt wird auf noch auf bei der Implementierung aufgetretene unvorhergesehene Schwierigkeiten eingegangen.

5.1 Architektur

Das Security-Plugin wurde in das bereits bestehende Prodynamics-Plugin für JIRA integriert. Backendseitig ist der Code vom diesem aber unabhängig. Dabei wurde das Package `security` angelegt, in dem sich die für die Analyse relevanten Klassen befinden. Die ursprüngliche Paketstruktur von Vulnerability Checker und Security Code Clone Detector wurde nicht verändert.

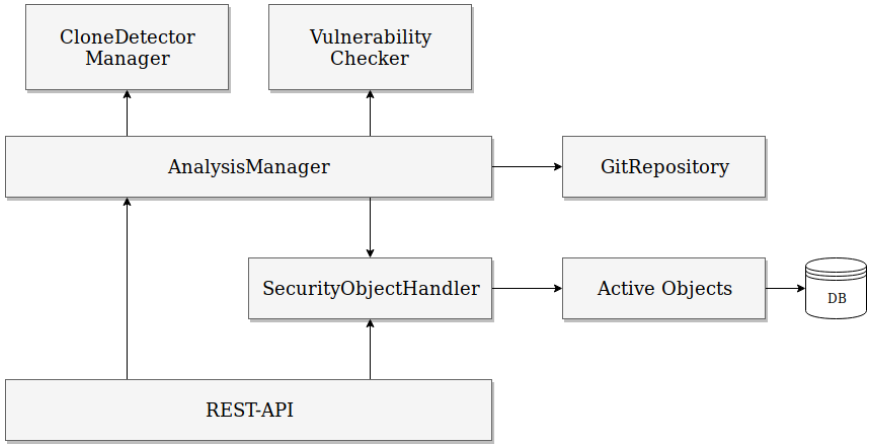

Abbildung 5.1: Schematische Darstellung Architektur Backend

Die schematische Struktur des Backends ist in Abb. 5.1 dargestellt. Dabei

bildet der `AnalysisManager` die zentrale Klasse des Plugins. Dieser bekommt Objekte zur Analyse über die REST-API reicht diese weiter und speichert die Ergebnisse. Darüber hinaus lässt sich sein Status zur Darstellung im Webinterface abfragen.

Die Klasse `CloneDetectorManager` bildet die Schnittstelle zwischen Security Code Clone Detector. Über die Methode `initialize` wird dieser initialisiert, mittels `preProcessSecurityRepository` das Security Code Repository eingelesen und durch `setSettingSourcererThreshold` ein vom Standardwert abweichender Threshold für den Detector gesetzt. `searchClones` startet die Clone Detection für die übergebene Source-Datei. Über die Attribute `resultDetails` und `displayData` lassen sich die Ergebnisse auslesen.

Analog dazu steht die Klasse `VulnerabilityChecker` für den Vulnerability Checker. Über `execute` initialisiert dieser seine NVD-Daten. Mittels `setConfiguration` setzt die Klasse `AnalysisManager` die Konfiguration für den Check und startet diesen über die Methode `getSearchJob`. Die Ergbnisse sind dann via `getVulnerabilityModel` abfragbar.

5.2 AnalysisManager

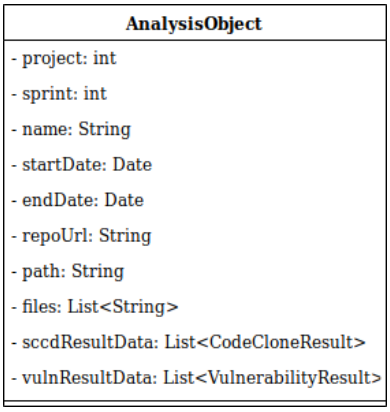

Abbildung 5.2: Klasse AnalysisObject (ohne Getter und Setter)

Der `AnalysisManager` implementiert ein Singleton und verwaltet in einer Warteschlange einzelne Analyseaufträge in Form von Objekten des Typs `AnalysisObject`. Er übernimmt die Initialisierung von Security Code Clone Detector und Vulnerability Checker. Hierbei wird auch das Security Code Repostory geklont bzw. aktualisiert und die aktuellen Daten der National Vulnerability Database heruntergeladen. Sobald ein `AnalysisObject` mittels Methode `add` der Warteschlange hinzugefügt wird, startet die Analyse wie

in Abb. 5.3 dargestellt: Der Stand des zum analysierten Projekt gehörigen Git-Repository wird auf den Zeitpunkt des Sprint-Endes gesetzt und die seit Sprint-Anfang geänderten Dateien ermittelt. Darauf folgt die Analyse des Quellcodes mittels Security Code Clone Detector und Vulnerability Checker. Danach werden die Ergebnisse in Form von Active Objects in der Datenbank gespeichert. Diese umfassen `SecurityAO` als Referenzobjekt mit Projekt– und Sprint-Daten, die im Verlauf des Sprints geänderten Dateien `SecurityFileAO` sowie die Ergebnisse von Security Code Clone Detector und Vulnerability Checker (`SecuritySccdAO` und `SecurityVcAO`). Schlussendlich wird das zugehörige Git-Repository wieder auf seinen Ausgangszustand zurückgesetzt. Dieser Vorgang wird wiederholt, bis die gesamte Warteschlange abgearbeitet ist.

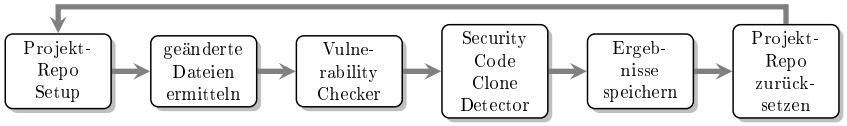

Abbildung 5.3: Schematischer Ablauf der Security-Analyse

5.3 Git-Integration

Alle Git-Funktionalitäten werden in der Klasse `GitRepository` gebündelt und mit der Bibliothek JGit[1] realisiert. Dabei handelt es sich um eine Java-Implementation von Git.

`GitRepository` abstrahiert die Funktionen der Bibliothek und stellt Methoden für wiederkehrende Abläufe bereit. Der Konstruktor nimmt einen lokalen Pfad oder die URL zu einem entfernten Repository entgegen und initialisiert die Klasse entsprechend. Wenn bereits ein lokales Repositorium existiert, wird dessen Stand mit dem entfernten Gegenpart abgeglichen. Besteht noch kein lokales, so wird das entfernte kopiert und mit dem neuen lokalen Repository weitergearbeitet.

Eine im Rahmen der Sprint-bezogenen Analyse wichtige Funktion stellt die implementierte Methode `getChangedFiles` dar. Damit nur die im jeweiligen Sprint geänderten Quelldateien eines Projektes in die Analyse einfließen, müssen diese ermittelt werden. Hierzu wird die durch die Versionskontrolle bereitgestellte Änderungshistorie verwendet. Mittels übergebenen Start– und Endzeitpunkt des Sprints, wird über alle dazwischen liegenden Änderungseinträge iteriert und die jeweiligen Änderungen einer Liste hinzugefügt. Dabei müssen entfernte und umbenannte Dateien speziell

[1] JGit: `https://www.eclipse.org/jgit/`

berücksichtigt werden. Anschließend wird die gesamte Liste an geänderten Dateien des Projekts zurückgegeben.

Für die Integrität der Projektquellen während der Analyse ist notwendig, dass sich der Versionsstand nicht während dieser ändern kann. Da die Auswertung der Daten auf Dateisystemebene stattfindet, kann das gleichzeitige Aktualisieren der Repository-Daten die Ergebnisse verfälschen. Deshalb findet der Zugriff auf die Klasse `GitRepository` ausschließlich über die Instanz des `AnalysisManager`s statt. Durch den iterativen Ablauf der Analyseschritte (vgl. Abb. 5.3) bleibt die Integrität der Daten währenddessen gewahrt.

5.4 REST-API

Die Schnittstellen sind in Form einer REST-API in der Klasse `ActiveObjectsRest` implementiert. Diese stellt hauptsächlich Daten in Form von Active Objects für Web-Frontend bereit. Darüber hinaus gibt es eine Schnittstelle für die Konfiguration, die nicht mit Active Objects arbeitet. Diese wird in Abschnitt 5.5.3 erläutert. Die REST-Schnittstelle umfasst folgende Endpunkte:

`/security/{project}/{sprint}/{name}/{repo_url}/{start_date}/{end_date}`
: nimmt die Metadaten des Sprints entgegen und erzeugt daraus ein `AnalysisObject` und übergibt dies den `AnalysisManager` über seine Methode `add`.

`/security/files/{project}/{sprint}`
: liefert eine Liste von geänderten Dateien für den gegebenen Sprint.

`/security/get/{project}/{sprint}`
: gibt eine Liste aller Befunde von SCCD und Vulnerability Checker für gegebenen Sprint zurück.

`/security/chart/{project}/{sprints}`
: nimmt eine kommaseparierte Liste von Sprint-IDs entgegen und erzeugt die Analysedaten in aufbereiteter Form für die Darstellung als Diagramm (s. Abschnitt 5.5).

`/security/status`
: liefert den momentanen Status des `AnalysisManager` in Form eines HTTP-Statuscodes zurück. Status 200 (OK) steht dabei für *bereit*, 202 (Accepted) für *beschäftigt*, also eine laufende Analyse oder Initialisierung, und 500 (Internal Server Error) weist auf einen internen Fehler hin.

`/security/status/latest`
: gibt eine Kurzzusammenfassung zur letzten abgeschlossenen Analyse zurück.

`/security/status/messsage`
: antwortet mit der letzten Statusnachricht der aktuellen Analyse.

`/security/status/progress`
: gibt Fortschritt der laufenden Analyse im Wertebereich von 0 bis 100 zurück.

Die Verbindung zur REST-Schnittstelle findet aus der Weboberfläche von JIRA mittels AJAX-Anfragen statt. Die Schnittstelle liefert die Daten als Listen von XML-Objekten aus, die wiederum als JavaScript-Objekte benutzt und in der grafischen Oberfläche (s. Abschnitt 5.5) dargestellt werden. Eine Ausnahme bilden die Endpunkte mit dem Pfad `/security/status/*`. Diese geben ausschließlich Daten in Form vom Strings zurück.

5.5 Frontend und Benutzeroberfläche

Den Einstiegspunkt für die Benutzeroberfläche bietet die Übersichtsseite des ProDynamics-Plugins in JIRA (s. Abb. 3.3). Über die Menüpunkte *Security Checker*, *Administratiton* und *Settings* sind jeweils das Ergebnis der Sicherheitsanalyse, der Start der Analyse für einen bestimmten Sprint und die Konfiguration des Security-Plugins erreichbar.

5.5.1 Analyseergebnis

Das Ergebnis der Analyse wird in einem Kreisdiagramm dargestellt (s. Abb. 5.4). Dieses wird mithilfe der JavaScript-Bibliothek zur Datenvisualisierung D3.js[2] implementiert. Die Ringe des Diagramms stehen für die abgeschlossenen Sprints des Projekts, wobei der äußerste die Daten des aktuellsten analysierten Sprints darstellt. Jede im Projekt verwendete Quellcode-Datei und Bibliothek wird entsprechend ihres schwerwiegendsten Analysebefundes wie in Tabelle 5.1 dargestellt eingestuft und farblich gekennzeichnet. Die Kategorisierung erfolgt dabei anhand des CVSS-v2.0-Wertes (vgl. Tabelle 2.1), sofern eine Zuordnung des Befundes zur NVD existiert. Bei Überfahren des entsprechenden Segmentes mit der Maus, werden weitere Informationen und absolute Zahlen zu betroffenen Dateien in der Mitte des Diagramms angezeigt.

Unterhalb des Diagramms schließt sich eine Tabelle mit allen Ergebnissen der Analyse an. Diese sind nach Bibliotheken und Quelldateien geteilt. Während für Bibliotheken die CVE-ID mit Link zum entsprechenden Eintrag

[2] D3.js: `https://d3js.org/`

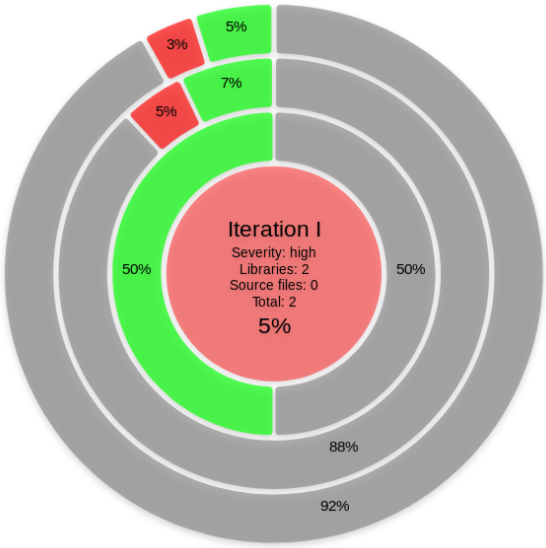

Abbildung 5.4: Diagrammdarstellung Analyseergebnis

	Farbe	Befunde	CVSS-Rating
	grün	keine	
	gelb	vorhanden	*Low*
	orange	vorhanden	*Medium*
	rot	vorhanden	*High*
	grau	vorhanden	nicht klassifiziert

Tabelle 5.1: CVSS *Kategorisierung der Dateien für die Diagrammdarstellung*

in der NVD und der zugehörigen CVSS-v2.0-Score gelistet wird, umfassen die Einträge für Quellcode zusätzlich Informationen zu den gefunden Clone-Pairs. Dies umfasst die Datei aus Security Repository, in der das Fragment gefunden wurde, sowie die zugehörigen Codezeilen und Namen der betreffenden Methodenname in beiden Dateien.

5.5.2 Administration

In der Administrationsansicht des ProDynamics-Plugins wird die Analyse nach Auswahl von Projekt und zugehörigem Sprint über den Button *Analyse starten* gestartet. Im Hintergrund wird via JavaScript über die JIRA-API das Start- und Enddatum des gewählten Sprints abgefragt und via Schnittstelle des Git-Plugins die URL des zugehörigen Repositoriums ermittelt. Diese Daten werden mit Betätigen des Buttons an die REST-Endpunkt `/security` des Security-Plugins übermittelt. Der entsprechende Sprint wird darauf in die Analysewarteschlange eingereiht.

Libraries

Filename	CVE ID	Score
trilead-ssh2-1.0.0-build221.jar	CVE-2011-0766	7.8
trilead-ssh2-1.0.0-build221.jar	CVE-2012-3478	2.1
mariadb-java-client-2.3.0.jar	CVE-2016-6664	6.9
mariadb-java-client-2.3.0.jar	CVE-2008-1106	7.1

Source Files

Filename	Function & line numbers	Filename of security match	Function of security match	CVE ID	Score
MainActivity.java	onCreateOptionsMenu 55-60	Stackoverflow40956487_1.java	DoGet 2-7		none
MainActivity.java	onClick 28-32	Stackoverflow16645937_1.java	hasXSSAttackOrSQLInjection 3-7		none
MainActivity.java	onCreate 19-43	Stackoverflow1265282_1.java	vulnerable 2-4		none

Abbildung 5.5: Tabelle Analyseergebnis

In Hintergrund wird in regelmäßigen Zeitabständen der Status der Analyse über die REST-Schnittstelle des Security-Plugins abgefragt. Der aktuelle Status wird farblich hervorgehoben neben dem Titel dargestellt. Sofern eine Analyse stattfindet, wird deren Verlauf mit einem Fortschrittsbalken visualisiert.

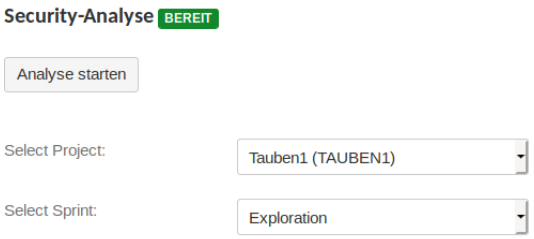

Abbildung 5.6: Auswahl von Projekt und Sprint zur Analyse

5.5.3 Konfiguration

Über den Menüpunkt *Settings* in der Weboberfläche des ProDynamics-JIRA-Plugins lassen sich die folgenden Einstellungen vornehmen: Git-Nutzername und Password, die URL zu Security Code Repository sowie der Pfad im Repository zu den für die Code-Clone-Suche aufbereiteten Daten. Die Daten für die Git-Authentifizierung werden dabei sowohl für Security Code Repository als auch für das Synchronisieren der zu analysierenden Projektrepositorien genutzt. Darüber hinaus besteht die Möglichkeit der Suchkonfiguration für Security Code Clone Detector und Vulnerability Checker jeweils in einer der drei Optionen *High Precision*, *High Recall* oder

Combined Precision and Recall. Die Konfiguration ist als HTML-Formular implementiert, das über den Button *Save* abgesendet wird.

```
1  private static final String PLUGIN_STORAGE_KEY =
   ↪  "de.unihannover.se.jira.plugin";
2  @ComponentImport
3  private final PluginSettingsFactory pluginSettingsFactory;
4
5  @Inject
6  public SettingsServlet(PluginSettingsFactory pluginSettingsFactory) {
7    this.pluginSettingsFactory = pluginSettingsFactory;
8  }
9
10 @Override
11 protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
12   Map<String, Object> context = Maps.newHashMap();
13   PluginSettings pluginSettings =
     ↪  pluginSettingsFactory.createGlobalSettings();
14
15   Object user = pluginSettings.get(PLUGIN_STORAGE_KEY +
     ↪  ".security.git.user");
16   Object password = pluginSettings.get(PLUGIN_STORAGE_KEY +
     ↪  ".security.git.password");
17
18   context.put("git-user", user);
19   context.put("git-password", password);
20
21   resp.setContentType("text/html;charset=utf-8");
22
23   templateRenderer.render("vm/dev/Settings.vm", context,
     ↪  resp.getWriter());
24 }
25
26 @Override
27 protected void doPost(HttpServletRequest req, HttpServletResponse
   ↪  response) {
28   PluginSettings pluginSettings =
     ↪  pluginSettingsFactory.createGlobalSettings();
29
30   String user = req.getParameter("git-user");
31   String password = req.getParameter("git-password");
32
33   pluginSettings.put(PLUGIN_STORAGE_KEY + ".security.git.user", user);
34   pluginSettings.put(PLUGIN_STORAGE_KEY + ".security.git.password",
     ↪  password);
35
36   response.sendRedirect("settings?" + req.getQueryString());
37 }
```

Listing 2: Konfiguration-Servlet

```
1  <div id="form">
2    <form id="admin" class="aui" action="" method="POST">
3      <div class="field-group">
4          <label for="git-user">Git username:</label>
5          <input type="text" id="git-user" name="git-user" class="text"
           ↪  value="$!git-user">
6      </div>
7      <div class="field-group">
8          <label for="git-password">Git password:</label>
9          <input type="password" id="git-password" name="git-password"
           ↪  class="text" value="$!git-password">
10     </div>
11     <div class="field-group">
12         <input type="submit" value="Save" class="button">
13     </div>
14   </form>
15 </div>
```

Listing 3: Velocity-Template des Formulars

Die zugehörige Logik im Java-Backend wird vom Servlet `SettingsServlet` übernommen. Die Speicherung der Konfiguration wird dort mit dem Interface `PluginSettingsFactory` des Plugin-Frameworks für JIRA realisiert. Dieses nutzt das *Abstract Factory Pattern*, um Objekte in Form von Key-Value-Paaren global für die JIRA-Instanz zu speichern [1]. Die Schnittstellen hierfür bilden `PluginSettingsFactory.get(key, value)` und `PluginSettingsFactory.set(key, value)`. Das `SettingsServlet` stellt die Methoden `doGet` und `doPost` für HTTP-GET- sowie POST-Requests bereit (vgl. Listing 2, Zeilen 11-24 und 27-37).

Beim GET wird erst die Berechtigung geprüft, danach die Konfigurationswerte geladen und schließlich dem Template-Renderer übergeben (Zeile 23, Listing 2). Dieser ersetzt die entsprechenden Variablen mit Präfix `$` im Velocity-Template `Settings.vm` (Listing 3, Zeilen 5 & 9) und rendert das ausgefüllte HTML-Formular. Falls noch keine Werte gesetzt wurden, wird dabei die Standardkonfiguration genutzt. Erhält das Servlet einen POST-Request, werden die Daten validiert und via `pluginSettingsFactory` gespeichert.

5.6 Unvorhergesehene Schwierigkeiten

Trotz der zuvor durchgeführten Tests ist es während der Implementierung zu unvorhergesehenen Schwierigkeiten gekommen. Während der Analyse kam es zu wiederkehrenden Abstürzen von JIRA, die von der Meldung „`java.lang.OutOfMemoryError: Java heap space`“ begleitet wurden. Der Fehler ließ sich beim Suchlauf einer großen Anzahl von Bibliotheken im

JIRA-Plugin reproduzieren. Bei Prüfung des Sachverhaltes im Vulnerability Checker mit dem gleichen Testset außerhalb von JIRA, traten jedoch keine Probleme auf.

Es wurde nach einem möglichen Memory-Leak durch eventuell nicht geschlossene Datenbankverbindungen gesucht, doch bei der regelmäßigen Garbage-Collection durch die Java-Virtual-Machine wurden alle Datenbankobjekte erwartungsgemäß aufgeräumt. Durch langwieriges Debuggen unter Beobachtung des Speicherverhaltens konnte das Problem schließlich gefunden und behoben werden: Der Vulnerability Checker stellt die Datenabfragen an die NVD-Datenbank sehr grob und bekommt pro Anfrage mehrere tausend Ergebnisse zurück, die anschließend auf relevante Ergebnisse gefiltert werden. Die Größe des Result-Sets übersteigt dabei den freien Anteil des allozierten Speichers der JIRA-Instanz. Diese Filterung wird nun direkt bei der Datenabfrage gefiltert, sodass das Ergebnis-Set nicht so groß werden kann, dass es signifikanten Einfluss auf den Speicherverbrauch bekommt.

Kapitel 6

Evaluation

6.1 Durchführung

Diese Evaluation analysiert in Form einer Studie insgesamt 84 Projekte aus den Jahren 2013 bis 2019. Die Datenbasis stellen die im Rahmen der Lehrveranstaltung Softwareprojekt am Fachgebiet Software Engineering der Leibniz Universität Hannover hervorgegangenen Arbeiten. Von den besagten Projekten, enthalten 65 Java- oder JAR-Dateien und gehen somit in das Ergebnis der Studie ein. Insgesamt umfasst die Analyse 8386 Java-Quelldateien und 992 Bibliotheken.

Als Basis für die Schwachstellenerkennung dient die National Vulnerability Database und das Security Code Repository mit einer kombinierten Datenbank aus von StackOverflow und Github exportierten Schwachstellen, die manuell von einem Security-Experten überprüft wurden. Beide Datenbanken besitzen zum Zeitpunkt der Durchführung den Stand vom 26. März 2019.

	Vulnerability Checker Konfiguration	Code Clone Detector Threshold
Konfiguration 1	High Recall	3.0
Konfiguration 2	Combined Precision & Recall	5.5
Konfiguration 3	High Precision	8.0

Tabelle 6.1: Konfiguration der Evaluationsdurchläufe

Die Suche wird in drei verschiedenen Konfigurationen durchgeführt. Diese unterscheiden sich bzgl. Recall und Precision beim Vulnerability Checker beziehungsweise Threshold beim Auffinden von Code Clones. Die Details der entsprechenden Konfigurationen sind in Tabelle 6.1 zu finden. Der Threshold ist der Schwellenwert, ab dem ein Code-Fragment als Clone erkannt wird. Ein höherer Wert erfordert eine höhere Ähnlichkeit [6]. Bei Recall und Precision handelt es sich um Maße zur Beurteilung der

Güte im Information Retrieval. Laut Wagner werden in der Konfiguration High Recall die meisten Schwachstellen gefunden, wobei es allerdings zu Falschmeldungen (False Positives) kommt. Dagegen ist High Precision treffsicherer, Schwachstellen können aber unentdeckt bleiben [18]. Weitere Informationen zur Konfiguration liefern die entsprechenden Masterarbeiten.

Die Projektquellen werden automatisiert durch ein Python-Skript via Subversion (Projekte bis Wintersemester 2016/17) bzw. Git kopiert und dann mit dem JIRA-Plugin von Security Code Clone Detector und Vulnerability Checker analysiert. Da die Projekte nicht in JIRA verwaltet werden, sind keine sprintbezogenen Informationen vorhanden. Somit wird jeweils der aktuelle Stand des gesamten Projekts ausgewertet. Für die automatisierte Analyse wird eine zusätzliche REST-Schnittstelle implementiert, die statt Sprintdaten den lokalen Pfad zu den Projektquellen entgegen nimmt und die Analyse startet. Ein Durchlauf aller Projekte dauert dabei rund 8 Stunden.

6.2 Auswertung

Die Gesamtergebnisse der Analyse sind in Tabelle 6.2 für Java-Bibliotheken und -Quellcode gegenübergestellt. Wie zu erwarten, nimmt die Anzahl der Funde mit höherer Precision bzw. Threshold ab. Während die Konfiguration auf die Ergebnisse des Vulnerability Checkers sichtbaren Einfluss hat, ist dieser beim Security Code Clone Detector überdeutlich erkennbar. Die durchschnittlichen Befunde pro Bibliothek lassen erkennen, dass verwundbaren Bibliotheken meist mehreren Schwachstellen zugeordnet sind.

	Java-Bibliotheken	**Java-Quellcode**
Dateien gesamt	992	8386
	Befunde absolut	
Konfiguration 1	4355	15713
Konfiguration 2	3561	99
Konfiguration 3	1957	28
	Anzahl Dateien mit Befund	
Konfiguration 1	229	4251
Konfiguration 2	149	84
Konfiguration 3	55	15
	Befunde relativ pro Datei	
Konfiguration 1	19,0	3,70
Konfiguration 2	23,9	1,18
Konfiguration 3	35,6	1,87

Tabelle 6.2: Gesamtergebnisse der Durchläufe

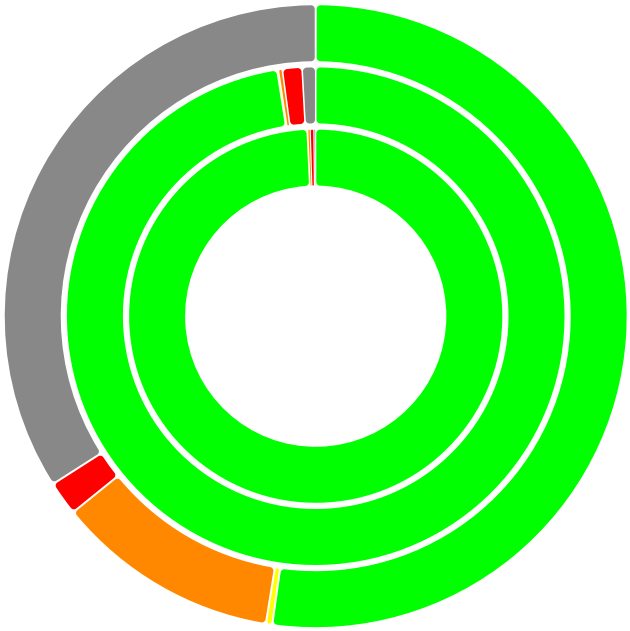

Abbildung 6.1: Befunde auf Dateien

		keine Schwachstelle	*low*	*medium*	*high*	unklassifiziert
Konf. 1	Quelldateien	4135	10	1048	0	3193
	Bibliotheken	763	21	42	166	0
Konf. 2	Quelldateien	8302	2	0	0	82
	Bibliotheken	838	6	24	119	0
Konf. 3	Quelldateien	8371	0	0	0	15
	Bibliotheken	937	6	20	29	0

Tabelle 6.3: Dateien nach Klassifizierung

Die Abb. 6.1 stellt die Befunde in Anlehnung an die Visualisierung des Security Plugins als Kreisdiagramm dar (vgl. Abschnitt 5.5.1). Jeder Ring zeigt die in einem Analysedurchlauf analysierten Dateien gruppiert nach ihrer jeweils schwersten Schwachstelle. Die Anordnung der Durchläufe erfolgt hier von außen nach innen, also Konfiguration 1 außen und Konfiguration 3 innen. Kategorisiert wird jede Datei anhand des schwerwiegendsten Befundes aufgrund des CVSS-Wertes der zugeordneten CVE-Daten. Dabei fällt auf, dass einem signifikanten Anteil der Dateien ausschließlich Schwachstellen ohne nach CVSS klassifizierten Befund zugeordnet werden (graue Farbe). Wie der Tabelle 6.3 zu entnehmen ist, betrifft dies nur von Code Clone Detector gefundene Schwachstellen im Java-Quellcode. Der Grund hierfür

ist der Datensatz des Security Code Repositorys. Im Folgenden wird deshalb getrennt auf Bibliotheken und Quellcode eingegangen.

6.2.1 Java-Quellcode

Die Basis für das Auffinden von Code-Clones bildet das Security Code Clone Repository und die darin vorhandenen Referenzen. Die Code-Fragmente stammen dabei von Github und StackOverflow, wobei nur die von Github einen Verweis auf die zugehörige CVE-ID besitzen. Ohne diese Verknüpfung kann keine Einstufung nach CVSS-Score erfolgen und die Schwachstellen bleiben diesbezüglich unklassifiziert.

Konfiguration 1		
CWE		Anzahl
CWE-295	Improper Certificate Validation	5914
CWE-310	Cryptographic Issues	4965
CWE-200	Information Leak	2246
Konfiguration 2		
CWE		Anzahl
CWE-79	XSS	82
CWE-295	Improper Certificate Validation	11
CWE-200	Information Leak	2
Konfiguration 3		
CWE		Anzahl
CWE-79	XSS	26
CWE-295	Improper Certificate Validation	1
CWE-352	CSRF	1

Tabelle 6.4: Anzahl Befunde gruppiert nach CWE, Höchstzahlen pro Durchlauf

Das Security Code Repository listet daneben allerdings die *Common Weakness Enumeration*, kurz CWE. Diese ist allen Datensätzen des Repositorys zugeordnet und kategorisiert die Art der Schwachstelle. Damit können die gefundenen Code-Clones dementsprechend gruppiert werden, um eine Übersicht über die häufigsten Arten aufgetretener Schwachstellen zu erhalten. Das Ergebnis in Tabelle 6.4 zeigt, dass über alle Durchläufe hinweg Schwachstellen der Kategorie *Improper Certificate Validation* zu den häufigsten Befunden zählen. Dabei ist fraglich, dass in den 65 analysierten Projekten überhaupt an 5914 Stellen Zertifikatprüfungen vorkommen. Die Anzahl nimmt in den folgenden Durchläufen mit höherem Threshold-Wert für die Clone-Detection sichtbar ab.

Im dritten Durchlauf mit einem Threshold von 8.0 bleibt nur ein einziger Befund dieser Kategorie übrig. Es handelt sich dabei um die Methode `enableSSLSocket` aus dem Projekt *volleyball1* des Wintersemesters

2016/17. Vergleicht man den Quellcode des Referenzfragmentes (Listing 4) mit der beanstandeten Datei (Listing 5), ist festzustellen, dass die Clone-Detection hier sehr gut funktioniert und der Code eine Schwachstelle in der Zertifikatsverifizierung für eine SSL-Session besitzt.

```
class Stackoverflow13076511_1{
void vulnerable(){
HostnameVerifier hv = new HostnameVerifier() {
    @Override
    public boolean verify(String arg0, SSLSession arg1)
    {
      // TODO Auto-generated method stub
      return true;
    }
  };
}
}
```

Listing 4: Stackoverflow13076511_1.java

```
public static void enableSSLSocket() throws KeyManagementException,
↪  NoSuchAlgorithmException {
  HttpsURLConnection.setDefaultHostnameVerifier(new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
      return true;
    }
  });
```

Listing 5: Auszug aus BeachNewsController.java, Projekt *volleyball1*

Tabelle 6.5 ordnet die gefundenen Code-Clones pro Durchlauf nach absoluter Häufigkeit. Betrachtet man beispielhaft den Quelltext des Code-Fragments mit der höchsten Trefferzahl in Listing 6, ist zu erkennen, dass das Fragment für sich genommen keine sicherheitsrelevante Aussagekraft besitzt. Es handelt sich dabei um eine Methode innerhalb einer Klasse, in der unsichere kryptografische Funktionen benutzt werden. Da die Tokenisierung des Security Code Clone Detectors aber auf Methodenebene arbeitet, geht dieser Kontext verloren. Die Erhöhung des Threshold-Wertes sorgt allerdings dafür dass diese Methode in folgenden Durchläufen nicht mehr berücksichtigt wurde, wie in Tabelle 6.5 zu erkennen ist. Bei längeren Fragmenten wie der Methode `doPost` aus der Datei `Stackoverflow385500_1.java` (Listing 7) ist dies jedoch auch bei einem Threshold von 8.0 noch der Fall.

Code-Fragment			Anz.
Datei	Zeilen	Methode	
Stackoverflow13735104_1.java	49–51	getN	4828
Stackoverflow13076511_1.java	4–9	verify	4423
GhCef44c7d305567c99....java	2–4	setDetails	2176
Stackoverflow385500_1.java	7–9	ImageDisplayServlet	53
Stackoverflow385500_1.java	145–147	doPost	26
Stackoverflow13076511_1.java	4–9	verify	9
Stackoverflow385500_1.java	145–147	doPost	26
Stackoverflow13076511_1.java	4–9	verify	1
Stackoverflow32600710_1.java	2–5	registerStompEndpoints	1

Tabelle 6.5: Die acht meistgefundenen Code Clones nach Konfiguration

```
1 public BigInteger getN() {
2   return n;
3 }
```

Listing 6: Auszug aus Stackoverflow13735104_1.java

```
1 protected void doPost(HttpServletRequest request, HttpServletResponse
  ↪  response) throws ServletException, IOException {
2   doGet(request, response);
3 }
```

Listing 7: Auszug aus Stackoverflow385500_1.java

Das Security Code Repository besitzt die Attribute `borderline` und `string_comparison`, die der Security Code Clone Detector bei der Analyse jedoch nicht berücksichtigt. `borderline` markiert die Einträge, die nur bedingt für die Code Clone Detection geeignet sind. Code-Fragmente, bei denen der Inhalt von Zeichenketten, die der Code Clone Detector bei der Tokenisierung nicht berücksichtigt, für die Schwachstelle relevant ist, sind mit `string_comparison` gekennzeichnet. Werden die Ergebnisse mithilfe dieser Attribute gefiltert, so erhält die in Tabelle 6.6 dargestellten Ergebnisse für die einzelnen Durchläufe.

Die oben angemerkten Negativbeispiele wurden durch diese Filterung komplett entfernt. Der einzige Befund, der in den Ergebnissen des Durchlaufs mit Konfiguration 3 nach dem Filtern übrig ist, ist die Schwachstelle der Zertifikatsprüfung im Projekt *volleyball1*. Die Daten zeigen, dass die Clone-

	Befunde	
	ungefiltert	gefiltert
Konfiguration 1	15713	6251
Konfiguration 2	99	13
Konfiguration 3	24	1

Tabelle 6.6: Ergebnis der Clone-Detection

Detection nur so gut sein kann, wie die Datenbasis, auf der sie stattfindet. Ohne die manuell redigierten Attribute zur Filterung, sind die Ergebnisse nur mit Einschränkungen verwertbar.

6.2.2 Java-Bibliotheken

Die Grundlage für die Analyse der Vulnerability Checkers bilden die Daten der National Vulnerability Database. Dadurch sind jedem Befund CVE- und CVSS-Daten zugeordnet, sodass sich die gefundenen Schwachstellen in den verwendeten Bibliotheken direkt miteinander vergleichen lassen.

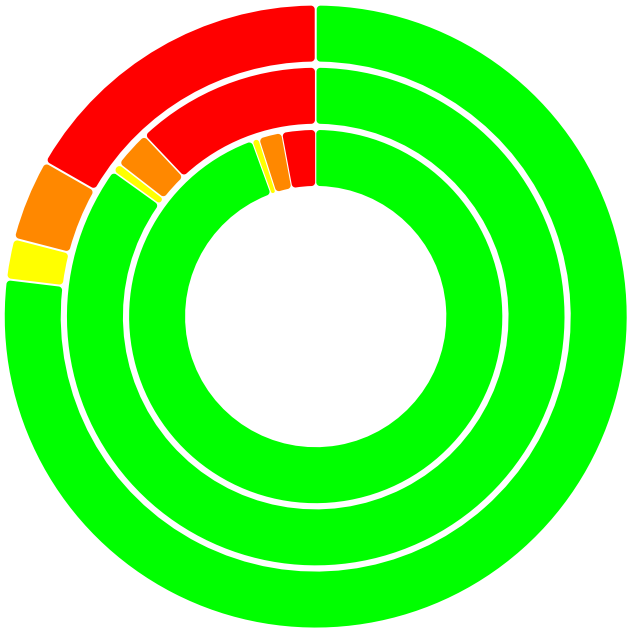

Abbildung 6.2: Befunde auf Bibliotheken

Abb. 6.2 stellt das Diagramm aus Abschnitt 6.2 analog nur für die Ergebnisse bezüglich der Java-Bibliotheken dar. Die Anzahl der Befunde nimmt mit jedem Durchlauf sichtbar ab. Während mit Konfiguration 1 (High Recall) Schwachstellen in 229 der 992 Bibliotheken gefunden werden, vermindert sich diese Zahl auf 55 Bibliotheken mit Schwachstellen im

	keine Schwachstelle	*low*	*medium*	*high*	unklassifiziert
Konfiguration 1	763	21	42	166	0
Konfiguration 2	838	6	24	119	0
Konfiguration 3	937	6	20	29	0

Tabelle 6.7: Bibliotheken nach Einstufung

Durchlauf mit der dritten Konfiguration (High Precision). Die zugehörige Tabelle 6.7 mit den absoluten Zahlen verdeutlicht, dass sich die Zahl der mit hoher Kritikalität eingestuften Bibliotheken, mit 82% zwischen ersten und letztem Durchlauf, besonders stark reduziert.

Es ist festzustellen, dass trotz des teilweise höheren Alters der Projekte, insgesamt je nach Konfiguration verhältnismäßig wenig bekannte Schwachstellen in den verwendeten Bibliotheken zu finden sind. So werden selbst im Durchlauf mit High-Recall-Konfiguration 77% der Bibliotheken als unkritisch gewertet. Dies bedeutet, bei einer Gesamtzahl von 4355 Befunden in dieser Konfiguration, eine Zahl von durchschnittlich 19 CVE-Einträge pro beanstandeter Bibliothek. Tabelle 6.8 stellt die Anzahl der verschiedenen CVEs pro Durchlauf dar, wobei doppelte Einträge zusammengefasst sind.

Konfiguration	Anzahl
Konfiguration 1	909
Konfiguration 2	757
Konfiguration 3	378

Tabelle 6.8: Zahl der verschiedenen CVEs nach Durchlauf

Da die Zahl gefundener Schwachstellen mit dem Alter von Software steigt, ist die Betrachtung des zeitlichen Verhältnisses zwischen Veröffentlichung der CVE und der Nutzung im Softwareprojekt interessant. Waren die Fehler bei Verwendung der Bibliothek schon bekannt oder wurden Schwachstellen erst danach entdeckt? Jeder Eintrag in der National Vulnerability Database besitzt Informationen zu seinem Veröffentlichungszeitpunk. Ordnet man jeder gefundene CVEs ihr Veröffentlichungsjahr zu, erhält man die Werte in Abb. 6.3. Diese wurde mit den Ergebnissen aus dem Analysedurchlauf mit High-Precision-Konfiguration erstellt, um den Einfluss von False Positives auf das Ergebnis zu minimieren.

Die Daten zeigen, dass Schwachstellen gefunden wurden, deren Veröffentlichung länger zurückreicht als das Jahr 2013, aus dem die ältesten Projekte stammen. Dies legt die Vermutung nahe, dass Bibliotheken zum jeweiligen Projektzeitpunkt bereits veraltet waren. Im Folgenden werden die Veröffentlichungsdaten der Schwachstellen mit den Beginn des Softwareprojekts

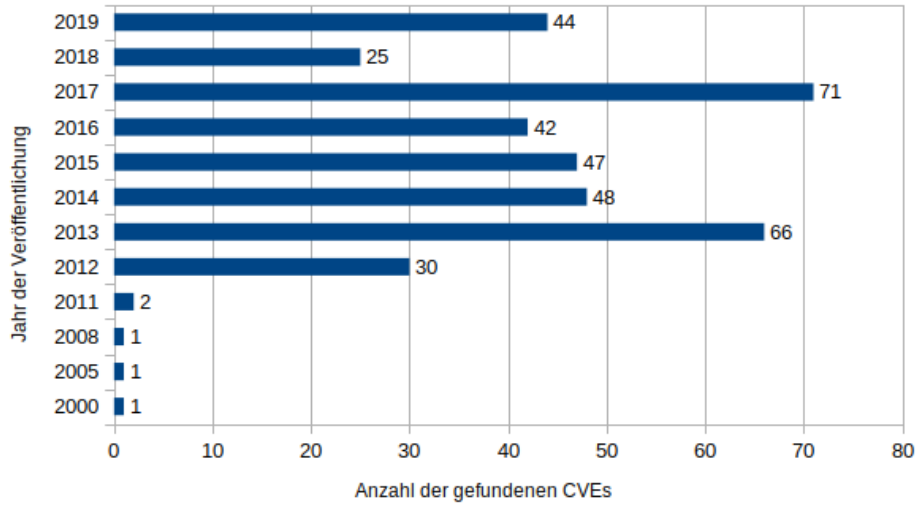

Abbildung 6.3: Gefundene CVEs nach Veröffentlichungsjahr, Konfiguration 3

verglichen. Da dieses jährlich im Wintersemester stattfindet, wird der 1. Oktober des entsprechenden Jahres als Stichtag angesetzt. Auch hier wurden die Daten vom dritten Durchlauf verwendet.

Die Ergebnisse in Tabelle 6.9 zeigen, dass auch Bibliotheken mit bereits bekannten Schwachstellen benutzt wurden. Dies stellt ein Sicherheitsrisiko dar, das vermeidbar wäre. Sofern der Entwickler automatisch über eine solche Schwachstelle informiert wird, hat dieser die Möglichkeit die ausgehende Risiken abzuwägen und die Bibliothek durch eine fehlerbereinigte Version zu ersetzen.

	Bibliothek mit Schwachstelle	
Jahrgang	bereits bekannt	noch unbekannt
2013/14	3	7
2014/15	3	10
2015/16	3	15
2016/17	1	1
2017/18	7	2
2018/19	2	1

Tabelle 6.9: Nutzung von Bibliotheken mit bereits bekannten Schwachstellen

Schaut man sich die älteste zugeordnete Schwachstelle mit der CVE-ID `CVE-2000-0863`[1] an, zeigt sich, dass man diese Ergebnisse nur als Richtwert betrachten kann. Die Bibliothek `jinput-platform-2.0.5-natives-linux.jar`, der die Schwachstelle

[1] NVD - CVE-2000-0863: `https://nvd.nist.gov/vuln/detail/CVE-2000-0863`

zugeordnet wurde, ist erst im März 2011 veröffentlicht worden. Es handelt sich somit um eine Falschzuordnung, die auf der Tatsache beruht, dass die JAR-Datei kein Manifest besitzt und somit allein der Dateiname zu Schwachstellensuche herangezogen werden kann. Trotzdem ist der Hinweis auf eine mögliche Schwachstelle, die keine ist, besser, als die Nutzung einer veralteten Version, die die Gefährdung von Informationssystem nach sich zieht. An Ende liegt es beim Entwickler zu entscheiden, wie er mit einer möglichen Schwachstelle umgeht. Entwicklungswerkzeuge können ihn dabei nur unterstützen.

Kapitel 7

Verwandte Arbeiten

Wie den Ausführungen zu verwandten Arbeiten in der Masterarbeit von Brunotte [6] zu entnehmen ist, existieren zahlreiche Tools zur Code-Clone-Detection. Die Studie *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach* von Roy et al. [15] vergleichen verschiedene Programme zum Auffinden von Code-Clones und deren Ansätze. Die Autoren legen vier definierte Szenarien fest, in denen Clones erkannt werden sollen. Für jedes Szenario werden Code-Fragmente künstlich nach speziellen Anforderungen erzeugt, um so die Grenzen der Clone-Erkennung einzelner Tools zu evaluieren. Die Autoren nennen das Auffinden von Bugs zwar als wichtigen Zweck der Clone-Detection, gehen in den erstellten Szenarien aber nicht weiter auf Softwaresicherheit ein. Die Analyse von Programmcode aus „echten“ Softwareprojekten findet nicht statt.

In dem Paper *Reliable Third-Party Library Detection in Android and its Security Applications* von Backes et al. [4] entwickeln die Autoren ein Programm zum Erkennen obfuskierter Java-Bibliotheken in Android-Applikationen und führen eine Studie deren Sicherheit durch. Dabei stellen sie fest, dass in fast drei Viertel aller Fälle veraltete Bibliotheken eingesetzt werden. Die Adaption neuer Versionen dauere dabei im Durchschnitt über 10 Monate.

Backes et al. analysieren einen Datensatz 4666 verschiedener Android-Apps, die über einem Zeitraum von 10 Monaten automatisch aus dem Google Play Store gecrawlt wurden. Dadurch lassen sich die Änderungen von Bibliotheken bei App-Updates in diesem Zeitraum nachvollziehen. Die Studie bezieht sich bei der Analyse der Bibliotheken nur darauf, wie aktuell die verwendete Version ist. Die gezielte Schwachstellensuche wird nur beispielhaft für zwei populäre Bibliotheken durchgeführt. Eine systematische Analyse potentieller Schwachstellen wie in dieser Arbeit findet nicht statt. Die Studie umfasst außerdem nicht dem eigentlichen Programmcode der Apps. Die Autoren beschränken sich bei der Auswertung auf verwendete

Bibliotheken.

Auf den Quellcode von Android-Apps bezieht sich dagegen *A Study of Android Application Security* [8]. Enck et al. führen darin eine statische Analyse über den dekompilierten Quellcode 1100 populärer Applikationen durch. Dabei wird auf Schwachstellen sowie auf missbräuchlich genutzte Funktionen eingegangen. Dadurch, dass der Quellcode nicht vorliegt und in einigen Fällen durch Dekompilieren nicht vollständig rekonstruierbar ist, beträgt die Abdeckung der Analyse nur knapp über 90% der Klassen. Bei der Studie kommt es somit zu *false negatives* – vorhandene kritische Funktionen können nicht gefunden werden.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Das im Rahmen dieser Arbeit entwickelte Security-Plugin für JIRA integriert Security Code Clone Detector und Vulnerability Checker und erweitert so die Analysemöglichkeiten der agilen Entwicklung mit JIRA. Durch die sprintbezogene Auswertung von Quellcode und Bibliotheken in Java-Projekten können Schwachstellen frühzeitig erkannt und deren Beseitigung in den iterativen Entwicklungsprozess integriert werden. Die Nutzer werden so im Bezug auf sicherheitskritische Fehler sensibilisiert.

Die Daten der Studie zeigen, dass es in studentischen Projekten zur Nutzung von Programmcode mit sicherheitskritischen Fehlern kommt und dass Java-Bibliotheken benutzt wurden, die bereits zum Verwendungszeitpunkt bekannte Schwachstellen besaßen. Dies kann durch die Integration von Analysetools für die automatische Fehlererkennung verbessert werden. Diese Tools stellen jedoch nur eine Hilfestellung für die Entwickler dar. Die Entscheidung, ob Fehler behoben werden oder eine Folgenabschätzung hinsichtlich der Relevanz gefundener Fehler, obliegt weiterhin der Verantwortung des Entwickler selbst.

Ein weiteres Ergebnis der Studie ist, dass das Resultat der Analyse nur so gut sein kann, wie die Datenbasis auf der sie beruht. Letztendlich ist doch oft das manuelle Redigieren von Datensätzen nötig, um möglichst präzise Fehlererkennung zu ermöglichen. Gerade bei der eingesetzten Clone-Detection auf Ebene von Methoden, müssen Referenz-Fragmente dafür geeignet sein. Darüber hinaus bedarf es einer regelmäßigen Pflege und Aktualisierung des Datenbestandes, um der fortschreitenden Entwicklung Sorge zu tragen.

8.2 Ausblick

Mit dem in dieser Arbeit entwickelten Werkzeugen können Statistiken zu Schwachstellen im Bezug auf Entwicklungs-Sprints analysiert werden. Die hierdurch gesammelten Daten lassen somit zukünftig feiner graduierte Studien zu, die wiederum mit anderen Daten aus JIRA und dem anderen Komponenten des ProDynamics-Plugins kombiniert werden können.

Hierbei könnten gefundene Schwachstellen mit anderen erfassten Projektdynamiken verknüpft und sprintbezogen visualisiert werden. Die Integration in JIRA kann erweitert werden. Eine Möglichkeit dazu wäre, automatisch Tickets mit hoher Priorität im Bugtracking zu erstellen, sobald es zu sicherheitskritischen Befunden kommt.

Die Integration von Vulnerability Checker und Security Code Clone Detector ist jeweils als Blackbox geschehen. Beide Programme werden jeweils über ihren zentralen Controller angesprochen, der alle internen Funktionalitäten steuert. Aufgrund ihres modularen Aufbaus wäre es allerdings möglich, diese Struktur weiter aufzuspalten und entsprechenden Programmteile direkt anzusteuern. Momentan ist der Download und die Aktualisierung der National Vulnerability Database fest integrierter Bestandteil des Vulnerability Checkers. Dieser wäre jedoch ohne hohen Aufwand zu entkoppeln. Wenn das JIRA-Plugin die entsprechenden Klassen direkt ansteuert, ist eine bessere Überwachung des aktuellen Status einzelner Komponenten und Parallelisierung von Abläufen möglich.

Die durch die Entkopplung höhere Kontrolle über die Komponenten würde es so möglich machen, momentan abstrahierte Daten zu nutzen, um in der Benutzeroberfläche gezeigten Informationen zu ergänzen. Funktionen wie das Logging der einzelnen Komponenten lassen sich in einer gemeinsamen Schnittstelle zusammenfassen, wodurch ein besseres Feedback dem Nutzer gegenüber möglich ist. In Kombination mit der von JIRA gebotenen Funktion der *Webhooks* können so direkte Nutzerbenachrichtigungen zu Ereignissen zu übermitteln werden, was ebenfalls das direkte Feedback gegenüber dem Nutzer und somit die Usability erhöht.

Literaturverzeichnis

[1] Atlassian SDK Resources - Storing plugin settings. https://developer.atlassian.com/server/framework/atlassian-sdk/storing-plugin-settings/. letzer Zugriff: 20.03.2019.

[2] National Vulnerability Database - Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss. letzer Zugriff: 18.03.2019.

[3] The Scrum Guide. https://www.scrumguides.org/scrum-guide.html. letzer Zugriff: 23.03.2019.

[4] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 356–367, New York, NY, USA, 2016. ACM.

[5] C. Bird, P. C. Rigby, E. T. Barr, , D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proceedings of the Sixth Working Conference on Mining Software Repositories*. IEEE Computer Society, May 2009.

[6] W. Brunotte. Security Code Clone Detection entwickelt als Eclipse Plugin. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2018.

[7] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. Spionage, Sabotage, Datendiebstahl: Deutscher Wirtschaft entsteht jährlich ein Schaden von 55 Milliarden Euro. https://www.bitkom.org/Presse/Presseinformation/Spionage-Sabotage-Datendiebstahl-Deutscher-Wirtschaft-entsteht-jaehrlich-ein-Schaden-von-55-Milliarden-Euro.html. letzer Zugriff: 11.03.2019.

[8] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[9] Y. Evers. Suche von sicherheitsrelevantem Code auf Stack Overflow. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2018.

[10] N. H. Pham, T. Nguyen, H. Nguyen, and T. N. Nguyen. Detecting recurring and similar software vulnerabilities. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 227–230, May 2010.

[11] Information technology – Security techniques – Information security risk management. Standard, International Organization for Standardization and International Electrotechnical Commission, 2018.

[12] J. Klünder, F. Kortum, T. Ziehm, and K. Schneider. Helping Teams to Help Themselves: An Industrial Case Study on Interdependencies During Sprints. In *Proceedings of HCSE2018: 7th International Working Conference on Human-Centered Software Engineering*. Springer, 2018.

[13] C. Müller. Security Code Exporter für Github. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2018.

[14] M. Pittenger. Open Source Security Analysis: The State of Open Source Security in Commercial Applications. Technical report, Black Duck Software, 2016.

[15] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009.

[16] A. Sheneamer and J. Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137:1–21, March 2016.

[17] G. Stoneburner, A. Goguen, and A. Feringa. Risk Management Guide for Information Technology Systems. Technical report, National Institute of Standards and Technology, 2002.

[18] L. E. Wagner. Konzept und Entwicklung eines Schwachstellenprüfers für Java-Bibliotheken. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2017.

[19] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 260–271, New York, NY, USA, 2015. ACM.