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

Heuristische Erkennung von Code-Exploits während der Programmausführung mithilfe von Call-Graphs

Heuristical Detection of Code-Exploits at Runtime using Call-Graphs

Bachelorarbeit

im Studiengang Informatik

von

Joshua Garlisch

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Dr. Jil Klüder
Betreuer: M. Sc. Fabien Patrick Viertel

Hannover, 03.08.2019

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 03.08.2019

_____

Joshua Garlisch

Zusammenfassung

Die Sicherheit von Software wird heutzutage immer wichtiger, da vor allem die Angriffe auf Computersysteme über die Jahre immer weiter zugenommen haben. Durch neue Gesetze werden Unternehmen immer weiter dazu bewegt personenbezogene Daten noch sicherer zu schützen. Alle Schwachstellen einer Software können nicht während der Entwicklungszeit entdeckt werden. Daher ist es relevant, Software auch zur Laufzeit zu überwachen. Bei verdächtigen Aktionen kann somit ein Entwickler alarmiert werden.

Aus den genannten Gründen wurde in dieser Bachelorarbeit ein Programm entworfen, das Code-Exploits zur Laufzeit von Java-Programmen erkennen soll. Dabei sollen mit Call-Graphs ein Modell erstellt werden, dass die Programme beschreibt, wie sie sich zur Programmausführung verhalten. Werden Unterschiede zur Programmausführung erkannt, ist zu prüfen, ob es sich bei der Abweichung um einen Exploit handelt. Wurden Code-Exploits gefunden, so sollen diese einem Entwickler gemeldet werden.

Auf diese Weise können Sicherheitslücken auch noch nach der Entwicklungs- und Testphase erkannt werden und es könnte somit in Zukunft zur zusätzlichen Sicherheit beitragen.

Abstract

Heuristical Detection of Code-Exploits at Runtime using Call-Graphs

The security of software is becoming increasingly important these days, as attacks on computer systems in particular have continued to increase over the years. New laws are driving companies to protect personal information more securely. All vulnerabilities of a software cannot be detected during the development time. Therefore it is relevant to monitor software at runtime. In case of suspicious actions, a developer can be alerted.

For these reasons, a program was designed in this bachelor thesis to detect code exploits during the runtime of Java programs. The aim is to use call graphs to create a model that describes the programs and how they behave during program execution. If differences to the program execution are detected, it must be checked whether the deviation is an exploit. If code exploits were found, they should be reported to a developer.

In this way, security vulnerabilities can be detected even after the development and testing phases, and it could contribute to additional security in the future.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Aufgrund der voranschreitenden Digitalisierung werden heutzutage viele Programme und Software-Bibliotheken entwickelt. Diese können im Umfang stark variieren und werden von einem oder mehreren Software-Entwicklern erstellt. Für das Erstellen von komplexer Software, werden meist mehrere Entwickler benötigt. Dies bedeutet, dass sich ein Entwickler alleine nicht mit der ganzen Software auskennt. Ein weiterer Grund dafür ist auch, dass die heutigen Programme viele Abhängigkeiten besitzen. Wenn eine Anwendung der Art komplex wird, kann es passieren, dass unbekannte Schwachstellen in der Software enthalten sind und von den Entwicklern während der Entwicklung nicht entdeckt werden. Zusätzlich gibt es noch Schwachstellen die zur Implementierungszeit nicht existiert haben. Darüber hinaus können mit in Verbindung von neuen Technologien oder neuen Angriffstrategien weitere Schwachstellen entstehen. Neben dem, dass weitere Schwachstellen entstehen können, steht auch das die letzten Jahre gezeigt haben, das Angriffe auf Computersysteme immer weiter zunehmen [2, 3].

Schwachstellen die zusätzlich von einem Angreifer ausgenutzt werden können, um beispielsweise Schadcode auszuführen heißen Exploits. Exploits die es dem Angreifer ermöglichen beliebigen Schadcode auszuführen, sind aus der Sicht der IT-Sicherheit, besonders gravierend und können fatale Auswirkungen auf das gesamte Systemumfeld haben. Dem Angreifer könnte es so gelingen, aus der Anwendung auszubrechen, und dann böswilligen Payload in das System nachzuladen. Auch sind andere Szenarien denkbar, wie zum Beispiel für Programme die personenbezogene Daten verarbeiten, die Daten zu entwenden oder gar zu verändern. Am 03.Mai 2019 hat Stefan Beiersmann einen Artikel [4] veröffentlicht in dem nach Aussage von Sicherheitsforschern zu dem Zeitpunkt bis zu 50.000 Unternehmen die SAP einsetzten potenziell vor Exploit-Angriffen unsicher waren. Für die Unternehmen kann das weitreichende Folgen haben, so können nach dem Artikel geschäftskritische Anwendungsdaten gelöscht werden, sowie das Stehlen von vertraulichen Daten ermöglichen.

1.1 Problemstellung

Ein Hauptgrund dafür ist der Software-Lebenszyklus. Programme werden in der Entstehung entwickelt und meistens umfangreich getestet. Später im Lebenszyklus kann es passieren, dass sich das System verändert. Dieses war in der Entwicklung jedoch nicht vorgesehen. Wenn ein Angreifer eine Schwachstelle oder Exploit gefunden hat und ausnutzt, wird dieses überwiegend erst spät vom Verteidiger entdeckt. Die Begründung dafür ist, dass der Verteidiger darüber meist nicht in Kenntnis gesetzt wird und von der Veränderung des Systems oder der Daten nichts mitbekommt.

1.2 Lösungsansatz

Eine Überlegung mit der sich diese Bachelorarbeit beschäftigt ist das Überwachen einer Anwendung während der Laufzeit mithilfe eines Runtime-Security-Monitors. Dieses hat den Vorteil das Schwachstellen, die erst zur Laufzeit auftreten, von dem Monitor entdeckt werden können. Um das Problem zu lösen, soll deshalb zuvor ein Modell von der zur überwachenden Anwendung erzeugt werden, das alle vorgesehene Programmabläufe beschreibt. Mit diesem Modell sollen dann die Methodenaufrufe, die während der Laufzeit aufgerufen werden, verglichen werden. Mittels einer heuristischen Erkennung soll schließlich erkannt werden, ob es sich um einen Exploit handelt.

1.3 Struktur der Arbeit

Zum Beginn dieser Arbeit werden in dem Kapitel 2 grundlegende Begriffe erklärt, die im Verlauf dieser Arbeit verwendet werden und zu einem Gesamtverständnis beitragen. Im Kapitel 3 Verwandte Arbeiten wird darüber eingegangen welche bestehende Arbeiten bereits existieren, die sich mit der Thematik auseinandersetzen. In dem Kapitel 4 Konzept wird das erarbeitete Konzept vorgestellt, sowie verschiedene Tracing Konzepte, die es ermöglichen Anwendungen während der Laufzeit zu protokollieren. In dem Kapitel 5 Implementierung wird über die Implementierung und Umsetzung des Runtime-Security-Monitors eingegangen, der während der Bachelorarbeit entstanden ist. Das Kapitel 6 Evaluation bewertet den Runtime-Security-Monitor, in dem er schwachstellenbehaftete Anwendungen überwacht und Code-Exploits erkennen soll. Abschließend wird in Kapitel 7 Zusammenfassung und Ausblick ein abschließendes Fazit gegeben und es werden zukünftige mögliche Arbeiten und Forschungen aufgezeigt.

Kapitel 2

Grundlagen

In den Grundlagen werden wichtige Themengebiete kurz erklärt, um eine Grundbasis zu schaffen. Das Kapitel behandelt und erklärt einen Call Graph, Java Bytecode sowie sicherheitstechnische Themen.

2.1 Call Graph

Ein Call Graph oder auch Aufrufgraph genannt, stellt die Aufrufe und Verbindungen von Methoden in einem Programm dar. So kann man anhand eines Call Graphs erkennen, welche Methode eine andere Methode aufruft. Da es sich bei Java um eine objektorientierte Programmiersprache handelt, kann man so in einem Call Graph die Kommunikation zwischen den einzelnen Klassen und den damit zur Laufzeit entstehenden Objekten erkennen. Ein Call Graph könnte exemplarisch wie folgt aussehen:

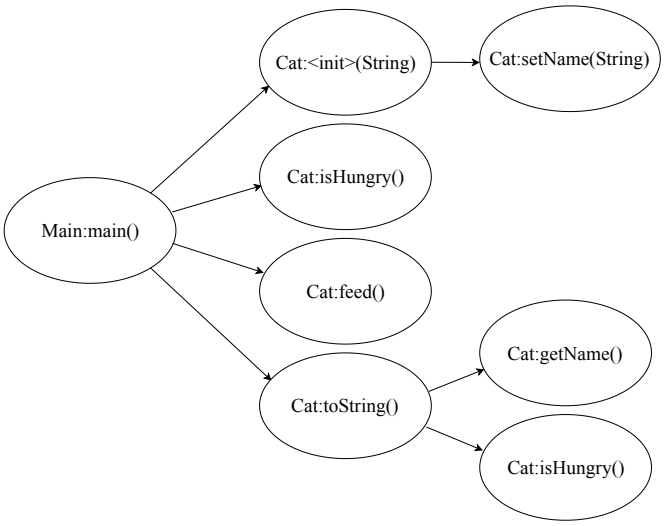

Abbildung 2.1: Call-Graph Beispiel

Dabei stellen die Ellipsen die Methoden einer Klasse dar und die Pfeile sind die Kantenbeziehungen, die den Aufruf einer Methode symbolisieren. Das Beispiel ist vereinfacht dargestellt, da für das Erzeugen von Objekten erst noch, wie es in Java üblich ist, die Oberklassen instanziiert werden müssen. Zur Übersichtlichkeit wurde darauf aber verzichtet.

2.2 Java Bytecode

Normalerweise wird in vielen Programmiersprachen der menschlich lesbare Quellcode vom Compiler in einen Maschinencode umgewandelt. Der Maschinencode kann dann von dem Prozessor ausgeführt werden. In Java wird der Quellcode nicht sofort in Maschinencode übersetzt, sondern erst in Java Bytecode, siehe Abbildung 2.2.

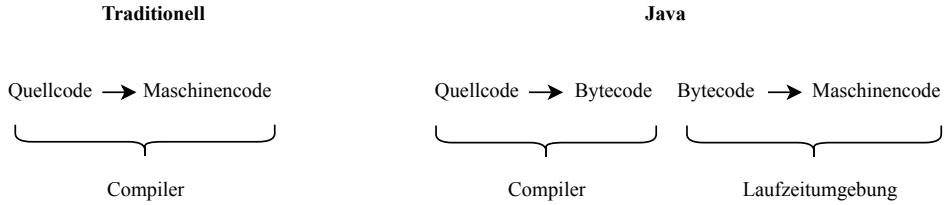

Abbildung 2.2: Grafik entnommen aus Java ist auch eine Insel [1]

Dateien mit Java Quellcode besitzen die Dateiendung ".java" und der Bytecode ".class". Der Java Bytecode ist deutlich kompakter, aber für den Menschen nicht mehr gut lesbar. Der Zwischenschritt von Bytecode ist deshalb sehr nützlich, da dieser von einer beliebigen Java Virtual Machine (JVM) interpretiert und damit plattformunabhängig ausgeführt werden kann. Der Befehlssatz der JVM in der SE 8 Edition enthält 204 Instruktionen, wie der JVM Spezifikation [5] zu entnehmen ist. Im Java Bytecode werden Konstanten in einem separaten Konstantenpool gespeichert. Eine Instruktion kann dann auf eine Konstante im Konstantenpool referenzieren, wenn diese von der Instruktion benötigt wird. Um Methodenaufrufe zu identifizieren gibt es in der Spezifikation [5] fünf verschiedene Instruktionen, siehe Tabelle 2.1 Instruktionen für den Methodenaufruf.

Instruktion	Beschreibung
invokedynamic	Eine Methode, die sich in dem Konstantenpool befindet, wird erst zur Laufzeit aufgelöst und aufgerufen.
invokeinterface	Eine Methode vom Interface wird aufgerufen.
invokespecial	Eine Methode der Instanz wird aufgerufen, mit der Besonderheit, dass es sich bei dem Aufruf um einen Initialisierungs-, Privaten Methoden- oder um einen Oberklassenaufruf handelt.
invokestatic	Eine statische Methode wird aufgerufen.
invokevirtual	Eine Methode von der Instanz wird aufgerufen.

Tabelle 2.1: Instruktionen für den Methodenaufruf

2.3 Exploit

In dem Kapitel soll der Unterschied zwischen einer Schwachstelle, eng. vulnerability, und einem Exploit aufgezeigt werden. Heutzutage gibt es eine Vielzahl von Schwachstellen. Dabei lassen sich ein Teil der Schwachstellen, mithilfe eines Exploits, ausnutzen. Der Exploit ändert dann den eigentlichen Programmablauf ab und führt stattdessen schädlichen Payload aus. Wie in dem Artikel [6] von Stefan Luber und Peter Schmitz beschrieben gibt es verschiedene Exploit-Arten:

- Remote ausgeführte Exploits
- Lokale ausgeführte Exploits
- Denial-of-Service-Exploits (DoS-Exploits)
- SQL-Injection-Exploits
- Command-Execution-Exploits
- Zero-Day-Exploits

Die Bachelorarbeit beschäftigt sich hauptsächlich mit den Remote ausgeführten Exploits, Command-Execution-Exploits und den Zero-Day-Exploits. Unter Remote ausgeführten Exploits versteht man das Ausnutzen von Schwachstellen aus der Ferne. So werden Datenpakete manipuliert und vom Server entgegen genommen und ausgewertet. Befindet sich in diesem manipulierten Datenpaket Programmcode und kann durch eine Sicherheitslücke zur Interpretierung und Ausführung kommen, so spricht man von einem Command-Execution-Exploit. Ein Command-Execution-Exploit ist sehr gefährlich da der Angreifer unter Umständen jeglichen Zugriff auf das komplette System erhält. Unter Zero-Day-Exploits versteht man die Exploits bei denen ein Unternehmen der die Software vertreibt erst am

Tag des bekannt werden über den Exploit informiert wird. Daher hat das betroffene Unternehmen keine Vorlaufzeit um einen Patch zu schreiben, der die ausnutzbare Schwachstelle behebt. Lokale ausgeführte Exploits und SQL-Injection-Exploits können beide nicht ohne Weiteres durch einen Call-Graph gefunden werden, da die beiden Arten entweder wie es bei den Lokalen Exploits der Fall ist, bereits in der Anwendung integriert sind oder bei den SQL-Injection-Exploits durch eine manipulierte SQL-Anweisung Daten in einer Datenbank verändert werden. Diese beiden Arten können aber selbst nicht durch eine Abweichung von der Programmausführung erkannt werden. Ein Exploit der ermöglicht Programmcode auszuführen, wird nachfolgend als Code-Exploit bezeichnet.

2.4 Deserialization

Ein wichtiger Teil in Java ist die Serialisierung. Durch das Serialisieren von Java Objekten wird ermöglicht, das die Objekte persistent gespeichert werden können oder zur Übertragung genutzt werden. Wenn ein Objekt serialisiert wurde und damit als Bytestrom vorliegt, lässt sich der Vorgang mit der Deserialisierung (eng. deserialization) rückgängig machen, damit wieder ein Java Objekt erzeugt wird. Jedoch ist dieses Verfahren sehr sicherheitsanfällig und wurde von der OWASP als Top 8 in den kritischsten Sicherheitsrisiken für Webanwendungen eingestuft [7]. Das Verfahren alleine ist nicht das Problem sondern viel mehr die Gefahren die hinter der Implementierung von Anwendungen und Bibliotheken stecken. Gerade wenn Benutzereingaben in Objekten umgewandelt werden und diese unzureichend geprüft wurden, kann es passieren das so Programmcode eingeschleust und ausgeführt wird. Auf Grund dessen kann dann die Anwendung vom eigentlichen Ablauf abkommen und zusätzlich böswilligen Code ausführen.

2.5 Statische und dynamische Analysen

In der IT-Sicherheit gibt es grundsätzlich zwei verschiedene Herangehensweisen um Programme auf Schwachstellen zu überprüfen. Es gibt eine Vielzahl an Vulnerability Scannern auf dem Markt, die in der Lage sind Schwachstellen zu identifizieren. Dabei Unterscheiden sich die Scanner in der Verfahrensweise wie sie arbeiten. Ein Teil der Programme untersuchen den Programmcode statisch auf Sicherheitslücken, in dem Sie die Code-Struktur analysieren. Eine andere Herangehensweise ist das Programm auszuführen und so mit anhand von Tests zu untersuchen wie das Programm reagiert. Beide Verfahren haben ihre Vor- und Nachteile, jedoch lässt sich grundsätzlich sagen das dynamische Analysen mit höheren Kosten in Bezug auf Zeit und Performance verbunden sind, als das es bei den statischen Programmanalysen der Fall ist. Eine Weitere Technik die sich von den beiden

Testverfahren abgrenzt, ist das verwenden von Runtime Application Self-Protection Software. Mit der Art von Software werden Programme erst während dem produktiven Einsatz überprüft. So kann das Tool dann wie von Margaret Rouse [8] beschrieben bei einem Aufruf von verdächtigen Funktionen informieren oder diese gar unterbinden.

Auch bei Viren Scannern wird zwischen statische und dynamische Scanverfahren unterschieden, um Maleware zu erkennen [9, 10]. Wie in dem Artikel [10] näher beschrieben, sind dynamische heuristische Scanner für kleinere Unternehmen und Privatanwender meist ungeeignet da Sie zu viel Zeit und Ressourcen benötigen.

Zusammenfassend lässt sich sagen, dass statische Verfahren sehr effizient im Aspekt auf Zeit und Performance sind, um Sicherheitsrisiken zu erkennen. Jedoch nicht als ausreichend zu betrachten sind, da diese nur so gut sind, wie die entsprechenden Datenbanken. Gerade dynamische Verfahren können sich gut eignen um Code-Exploits zu erkennen, jedoch haben dynamische Verfahren die Gefahr, dass die Laufzeit meist stark negativ beeinflusst wird.

Kapitel 3

Verwandte Arbeiten

In dem Kapitel Verwandte Arbeiten wird die Bachelorarbeit mit anderen Arbeiten verglichen und die Unterschiede aufgezeigt. So existiert von William G.J. Halfond und Alessandro Orso ein Paper *Combining Static Analysis and Runtime Monitoring* [11] in dem die Autoren ein Tool vorstellen mit dem versucht wird SQL-Injection[1] Attacken zu erkennen. Es wurde ebenso versucht ein statisches Modell zu erzeugen, umso die erwarteten legitimen SQL-Anweisungen zu erkennen. Das Tool überwacht dann die in der Laufzeit generierten Abfragen und prüft ob diese mit dem Modell übereinstimmen. Falls dieses nicht der Fall ist wird davon ausgegangen, dass es sich bei den Anweisungen um eine SQL-Injection handelt. Das Tool führt die potenziell gefährlichen Anweisungen nicht aus und meldet diese einem Entwickler. Für die Evaluation wurde das Tool mit zwei selbst erstellten kleinen Webanwendungen getestet und es wurden dabei alle SQL-Injections erkannt. Es wurden dabei auch alle legitimen Anweisungen ausgeführt und wurden nicht blockiert. Daher ist die Bewertung nach den Autoren positiv, machen aber zusätzlich darauf aufmerksam, dass die Testversuche nur auf kleinen Anwendungen basieren und das Programm nicht mit mittleren bis große Anwendungen getestet worden ist. Außerdem könnte es passieren das Probleme in der statischen Analyse auftreten, wenn Programm Konstrukte verwendet werden, wie der massive Einsatz von Reflexion. Zusätzlich könnten in der zu analysierenden Anwendung Blackbox-Komponenten enthalten sein, die das Programm nicht analysieren kann. Diese beiden Probleme könnten potenziell auch bei dem Runtime-Security-Monitor auftreten.

Eine andere Arbeit von Hajime Inoue und Stephanie Forrest mit dem Titel *Anomaly intrusion detection in dynamic execution environments* [12] stellt einen Ansatz vor, um eine Erkennung von Anomalien in dynamischen Laufzeitumgebungen durchzuführen. Dabei wurde ein Programm normal

[1] SQL-Injection bedeutet das ein Angreifer durch das Ausnutzen einer Schwachstelle, SQL-Anweisungen manipuliert und so diese vom eigentlichen Nutzen abändert. Dadurch ermöglicht das dem Angreifer die Daten in der Datenbank einzusehen oder diese abzuändern.

in der JVM ausgeführt und während der Programmausführung ein Profil erstellt. Anschließend wurde der Vorgang wiederholt, nur dieses mal wurde ein Exploit ausgeführt. Durch das zuvor erstellte Profil konnte der Exploit erkannt werden. Der Unterschied zu dieser vorliegenden Arbeit ist es, dass der Runtime Security Monitor vor der Ausführung ein Modell erzeugt und damit den Exploit erkennen soll. Der Vorteil ist der, dass die Anwendung nicht zuvor ausgeführt werden muss, um den Exploit zu erkennen.

Darüber hinaus gibt es weitere kommerzielle Programme wie Sqreen[2], die Webanwendungen zur Laufzeit im produktiven Einsatz überwachen. Die Produkte bieten meist ein breites Spektrum an Gefahrenabwehr an und melden Sicherheitslücken den Entwicklern. Ein Nachteil hinter diesen Programmen sind die verbundenen Kosten, sowie das kein genaues Wissen über die Funktionsweisen bekannt sind, da diese meist nicht veröffentlicht werden.

[2] `https://www.sqreen.com/`

Kapitel 4

Konzept

In dem Kapitel Konzept soll das Konzept hinter dem Runtime-Security-Monitor vorgestellt und erläutert werden. Mithilfe dieser Anwendung soll es möglich sein ein Java-Programm zu überwachen, um einen Code-Exploit zu erkennen. Es werden in dem Kapitel die Ziele behandelt, sowie den Programmablauf und die damit verbundenen Phasen, die für das Programm notwendig sind, um einen Code-Exploit zu bestimmen.

4.1 Ziele

Zum einen ist das primäre Ziel des Runtime-Security-Monitors das Entdecken von schwachstellenbehafteten Methoden, die durch einen Exploit genutzt werden. Darüber hinaus ist es sinnvoll herauszufinden, wie die schwachstellenbehaftete Methode aufgerufen worden ist. Ein sekundäres Ziel ist daher einem Entwickler den Programmablauf zur Verfügung zu stellen, wie die Schwachstelle aufgetreten ist.

4.2 Programmablauf

In dem Kapitel 4.2 soll das Konzept des Runtime-Security-Monitors erläutert werden.

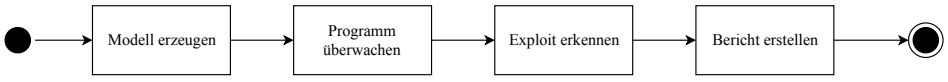

Abbildung 4.1: Grobes Ablaufkonzept

Die Abbildung 4.1 zeigt ein grobes Ablaufkonzept, in der die einzelnen Phasen zusehen sind, auf die in den Unterkapiteln näher eingegangen werden.

4.2.1 Modellerzeugung

In der Phase Modellerzeugung soll ein Modell ausgehend von dem zu analysierenden Programm erstellt werden, dass das Verhalten und die interne Kommunikation beschreibt. Mithilfe von einem Abstrakten Syntaxbaum, kurz AST, kann ein Quellcode auf die syntaktischen Bestandteile zerlegt werden. Jedoch lässt sich in einem AST, wie von Nicholas Smith et al. in *JavaParser: Visited* [13] beschrieben, nicht ohne Weiteres die einzelnen Referenzen zu den Methodenaufrufen abbilden. Ein Modell, das sich besser eignet, ist ein Call-Graph, da dieser die einzelnen Aufrufe der Methoden eines Programms beschreibt. Der Call-Graph kann dann als Modell dafür genutzt werden, um mögliche Methodensequenzen mit diesem Modell auf Validität zu überprüfen.

Ein vollständiger Call-Graph zu einem Programm würde alle möglichen Methodenaufrufe beinhalten. Bei großen Programmen erweist sich dieses aber als schwierig. Das Paper von Michael Reif et al. *Systematic evaluation of the unsoundness of call graph construction algorithms for Java* [14] evaluiert Bibliotheken und deren Algorithmen, die in Java einen Call Graph konstruieren. Es wird gezielt darauf eingegangen, ob die Bibliotheken den gesamten Call Graph vom ausgehend übergebenen Programmcode korrekt erstellt und wenn bei welchen Sprachfunktionen Kanten zu entsprechenden Methoden fehlen. Im Paper wird verdeutlicht das von den zwei vorgestellten Bibliotheken, keine Implementierung alle Tests erfolgreich besteht. Die Autoren stellen dar das vor allem neue Sprachfunktionen die seit Java 8 eingeführt wurden noch nicht ausreichend gut abgebildet werden.

Wie in den Grundlagen 2.2 Java Bytecode beschrieben, wird der Quellcode von dem Compiler erst noch in Bytecode übersetzt. Theoretisch könnten beide Formen zu der Erstellung von dem Modell genutzt werden. Für das Konzept hat sich aber der Java Bytecode durchgesetzt, da er für die Modellerzeugung drei grundlegende Vorteile aufweist. Zum einen spricht für den Bytecode, dass dieser einfacher zu interpretieren ist, da er mit dem JVM-Befehlssatz streng spezifiziert wurde und deutlich kompakter ist als der Quellcode. Außerdem gibt es weitere Programmiersprachen wie Scala und Co., die auf der JVM ausgeführt werden. Da diese Programmiersprachen ebenfalls auf dem Java Bytecode basieren, lassen sich die Programmiersprachen auch mit der Variante verwenden. Der wichtigste Hauptgrund ist aber der, dass manchmal nur der Bytecode zur Verfügung steht. Es gibt viele kommerzielle Projekte oder Bibliotheken, die nur den Bytecode bereitstellen und den Quellcode nicht herausgeben.

Zusammenfassend soll der Runtime Security Monitor eine JAR-Datei entgegennehmen und davon ausgehend einen Call-Graph erstellen, der die interne Kommunikation der einzelnen Methodenaufrufe beschreibt.

4.2.2 Tracing

Während der Konzeptionsphase wurden drei verschiedene Möglichkeiten gefunden, die sich dafür eignen laufende Java-Programme zu überwachen. Um eine Entscheidung zu treffen welche Variante sich am besten eignet wurde ein Kriterienkatalog erstellt.

Eine Möglichkeit ist dieses mithilfe der Java Platform Debugger Architecture, kurz JPDA, durchzuführen. Die JPDA bietet neben dem Programmverlauf noch zusätzliche Informationen, wie aktuelle Objektattribute. Doch hat sich während der Konzeptionsphase herausgestellt, das JPDA für das konkrete Vorhaben eine große Anzahl an Methodenaufrufe zu tracen nicht performant genug ist. Es wurde dafür eine kleine Beispielanwendung aufgezeichnet, bei der JPDA im Schnitt bis zu fünf mal langsamer war als die anderen Varianten. Eine Weitere Möglichkeit ist das Kieker Framework zu verwenden, das auf AspectJ aufbaut. Nachteil der Variante von dem Kieker Framework mit in Verbindung von AspectJ ist, dass es bei dem Weben von Klassen zur Ladezeit nicht möglich ist die *java.** bezogenen Klassen aufzuzeichnen [15]. Es ist aber von hoher Bedeutung auch Methoden zu verfolgen, die sich in der Java Runtime Environment (JRE) befinden, da gerade in der Methoden auf einen möglichen Code-Exploit überprüft werden können. Die dritte Variante ist mithilfe eines Java-Agents, das laufende Programm mit dem Instrumentation Interface von Java zu überwachen. So kann durch die Instrumentation der Java Bytecode manipuliert werden, bevor dieser in die JVM geladen wird. Mit der Manipulation von Java Bytecode ist es dadurch möglich den Bytecode entsprechend zu erweitern, sodass bei jedem Methodenaufruf Ein- und Austritt eine Klasse im Agenten benachrichtigt wird. Da über die Instrumentation eine bessere Performance ermöglicht wird, JRE-Methoden mit protokolliert werden können und sich durch die Bytecode Manipulation zukünftig auch noch weitere Funktionen hinzufügen lassen, wurde sich für die Instrumentation Schnittstelle entschieden.

Kriterien	**JPDA**	**Kieker Framework**	**Instrumentation**
Performance		X	X
JRE-Methoden	X		X
Bytecode Manipulation		X	X

Tabelle 4.1: Kriterienkatalog

4.2.3 Heuristische Erkennung von Schwachstellen

In der Phase wird ein Konzept vorgestellt, das ermöglicht mit dem Modell eines Call-Graphs und die protokollierten Methodenaufrufe eine Differenzenbildung durchzuführen, umso einen Exploit mit den ausgenutzten schwachstellen behafteten Methoden zu erkennen. Darüber hinaus muss bewertet werden ob es sich bei der erkannten Abweichung tatsächlich um einen Code-Exploit handelt. Im Grunde kann ab dem Zeitpunkt bei der die main-Methode aufgerufen wurde mit dem Call-Graph abgeglichen werden, welche Methode erlaubt ist aufzurufen und welcher Methodenaufruf nicht im Call-Graph enthalten ist. Um das Verfahren zu verdeutlichen wurde ein Quellcode 4.1 erstellt. So darf die main-Methode zur Laufzeit nur den Konstruktor der Klasse *Cat* aufrufen, sowie die Methoden *isHungry*, *feed*, *toString* und *System.out.println*. Wenn jetzt zur Laufzeit aber innerhalb der main-Methode *Runtime.getRuntime().exec("calc.exe");* aufgerufen wird, wäre dieses eine Abweichung und könnte durch einen Code-Exploit verursacht worden sein.

```
1  public static void main(String[] args) {
2    Cat cat = new Cat("Miez");
3
4    if(cat.isHungry()) {
5      cat.feed();
6    }
7
8    System.out.println(cat.toString());
9
10 }
```

Quellcode 4.1: Beispiel Methode

Da jedoch das Beispiel 4.1 stark vereinfacht ist und es in der Hochsprache Java eine Reihe an Sprachfunktionen gibt, ist eine eindeutige Bestimmung nicht immer so einfach möglich. Daher gibt es Grenzfälle bei denen zwar eine Abweichung zu erkennen ist, jedoch dieses darauf zurückzuführen ist, dass der Call Graph statisch ist und sich das ausgeführte Programm zur Laufzeit dynamisch verhält. In Java existiert zum Beispiel die Möglichkeit mithilfe der Reflection API dynamisch zur Laufzeit die Programmausführung zu ändern. Yue Li et. al. erläutern Java Reflection in ihrem Artikel [16] detailliert. Im Folgenden wird Reflection mit einem Quellcode Beispiel 4.2 grob erklärt und auf die Schwierigkeiten der Differenzenbildung mit einem statischen Modell, siehe Abbildung 4.2, aufgezeigt.

Um das Beispiel anschaulich zu verdeutlichen wurde eine Klasse *HelloWorld* mit der Methode *hello* erstellt. In der main-Methode wird daraufhin dynamisch zur Laufzeit die Klasse *HelloWorld* instanziiert und die Methode *hello* aufgerufen. Zum Schluss wird so dann auf der Konsole "HelloWorld!" angezeigt.

```java
public class Main {
  public static void main(String args[]) throws Exception {
    String cName = "HelloWorld";
    String mName = "hello";

    Class clz = Class.forName(cName);
    Object obj = clz.newInstance();
    Method mtd = clz.getDeclaredMethod(mName);
    System.out.println(mtd.invoke(obj));
  }
}

class HelloWorld{
  public String hello() {
    return "HelloWorld!";
  }
}
```

Quellcode 4.2: Beispiel Reflection

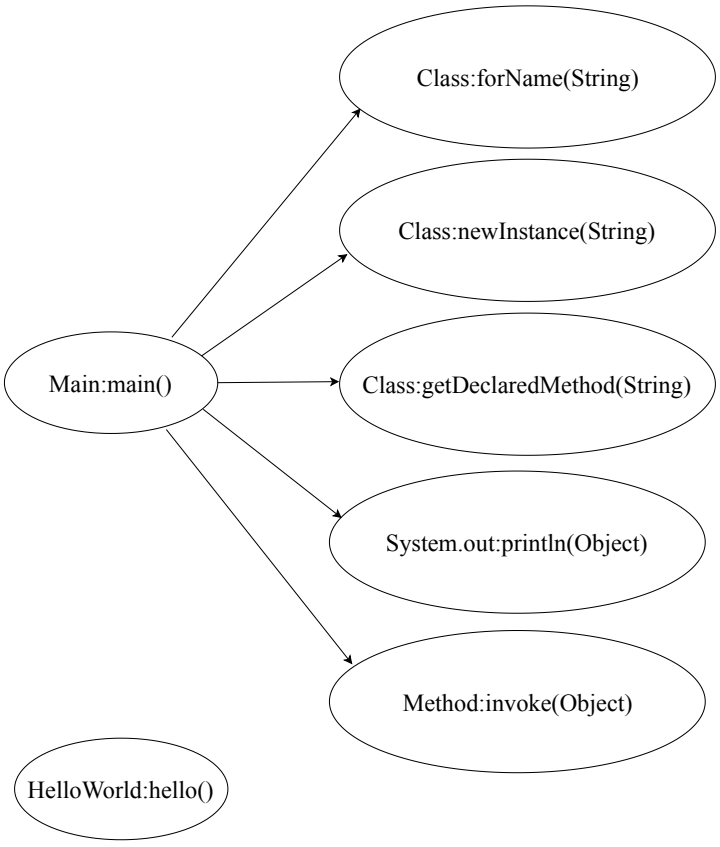

Abbildung 4.2: Call-Graph Beispiel Reflection

Die Variablen *cName* und *mName* könnten zu dem auch über die Konsoleneingabe übergeben werden. Da vor der Programmausführung ein statisches Modell erstellt wird, sind zwar die Methodenaufrufe von der Reflection vorhanden, jedoch aber nicht der konkrete Aufruf von der Methode *hello*. Yue Li et. al. betonten in ihrem Artikel [16] das die Forschungsarbeiten aus den vergangenen Jahren Java Reflection meist als separates Thema betrachten und statische Analysewerkzeuge Reflection nur teilweise oder gar nicht behandeln.

Das einfache Beispiel im Bezug auf Reflection zeigt deutlich das es eine Differenz zwischen Modell und der zur Laufzeit aufgerufenen Methoden gibt. Jedoch lässt sich direkt keine eindeutige Aussage darüber treffen, ob es sich bei der entstandenen Abweichung, um eine von dem Entwickler zur Entwicklungszeit als zulässige Abweichung verhält oder ob bei der Abweichung ein Code-Exploit vorliegt. In beiden Fällen sollte jedoch die Abweichung zunächst als mögliche Schwachstelle klassifiziert und verfolgt werden, da nicht klar ist ob es sich um eine böswillige Ausführung handelt.

Um eine präzisere Aussage darüber zu treffen, wurden drei zusätzliche Heuristische Verfahren entworfen, die das Laufzeit wissen nutzen, umso eine Differenz von Code-Exploit und regulärer Abweichung zu bilden:

1. Modell durch dynamische Programmausführung erweitern
2. Risiko Bewertung einzelner Methoden
3. Betrachtung von Parameterwerten

Modell durch dynamische Programmausführung erweitern

Ein möglicher Ansatz einen Großteil der regulären Abweichungen als unbedenklich zu bewerten ist, das diese Differenz zuvor mithilfe eines Programmdurchlaufes ermittelt wurden. Da diese Abweichungen als unbedenklich klassifiziert werden, sollte die Programmausführung in einer sicheren Umgebung durchgeführt werden, da sonst bei potenziellen Angriffen diese auch mit in das Modell aufgenommen werden würden. Die Programmdurchläufe werden dann zum regulären Start eingelesen, umso den Call-Graph mit zusätzlichen Kantenbeziehungen der einzelnen Methodenaufrufe zu erweitern.

Risiko Bewertung einzelner Methoden

Ein wichtiger Anhaltspunkt ist es zu erkennen, welche Methoden nach einer Abweichung aufgerufen werden. Handelt es sich bei den Methodenaufrufen um gefährliche Methoden die bekannt dafür sind, dass Exploits diese verwenden, so kann eine bessere Einordnung ermöglicht werden. Dazu werden zwei Sammlungen aus Methoden verwendet, es soll aber auch möglich sein dem Runtime-Security-Monitor eigene Auflistungen an gefährlichen Methoden zu übergeben. Eine Liste ist von OWASP

[17] und beinhaltet Methoden, die bekannt für Code Execution sind. Eine weitere Liste stammt aus dem Paper *Automated Exploit Detection using Path Profiling* [18]. Die Autoren George Stergiopoulos et al. haben in dem Paper ein Tool namens Entroine vorgestellt. Das Tool erkennt durch statische Quellcodeanalyse automatisch Exploits und um das zu realisieren wurde eine Liste aus Methoden verwendet, bei denen es sich um Exploitable Methods handelt. Die Autoren haben die Methoden zusätzlich mit einer Metrik von einem Risiko von 1 - 5 bewertet. Befindet sich die Methode in der Liste so ist davon auszugehen, dass die Wahrscheinlichkeit stark erhöht ist, das es sich um einen Exploit handelt.

Betrachtung von Parameterwerten

Ein weiteres Verfahren für die Erkennung von einem Exploit sollen Rückschlüsse auf enthaltene Werte in Parametern sein. So würde bei einer Abweichung bei denen die Parameterwerte zum Beispiel "System32" oder ".exe" enthalten, diese Abweichungen stärker bewertet werden, dass es sich um einen Exploit handeln könnte. Auf der anderen Seite ist zu erwähnen, dass gerade dieses Verfahren sehr subjektiv sein kann, da gezielt die Auswahl der vordefinierten Parameterwerte eine entscheidene Rolle spielt. Um eine bessere Differenzierung über die Gefährlichkeit der Werte zu erhalten, soll wie bei der Risikobewertung von den Methodennamen eine Skala von 1 - 5 verwendet werden. Neben dem Vergleichen von Zeichenketten sollen auch auf reguläre Ausdrücke überprüft werden können.

Die einzelnen Verfahren sollen miteinander kombinierbar sein. So soll ermöglicht werden das im ersten Schritt das statische Modell mit dynamischen Methodenaufrufen erweitert werden kann, sowie eine Bewertung von gefährlichen Methoden und Parameterwerten durchzuführen. Um die Methodennamen und Parameterwerte miteinander zu kombinieren könnte das harmonische Mittel oder das arithmetische Mittel verwendet werden. Bei dem harmonischen Mittel wird eine Abweichung dann als Exploit eingestuft, wenn beide Faktoren hoch sind. Ist dabei aber ein Faktor besonders niedrig zum Beispiel eins und der andere Faktor fünf, so ist das errechnete Gesamtrisiko nur 1,6. Gerade wenn einer der Faktoren hoch ist sollte dieser nicht so stark von dem niedriegeren Faktor beeinflusst werden. Berechnet man Anstelle vom harmonischen Mittel das arithmetische Mittel so erhält man ein Gesamtrisiko von drei. Daher wurde sich für das arithmetische Mittel entschieden. Mit einem threshold (englisch für Schwellenwert) kann dann zusätzlich ein Grenzwert defeniert werden, ab welchen Gesamtrisiko von einem Exploit auszugehen ist. Würde man den threshold auf drei defenieren, so wären alle Abweichungen mit einem Gesamtrisikofaktor von drei oder höher ein Exploit.

4.2.4 Bericht und Speicherung

In der letzten Phase müssen die gefundenen Exploits in Form eines Berichts gespeichert werden, sodass ein Entwickler oder eine weitere Software die möglichen Befunde weiterverarbeiten kann. Aufgrund der Tatsache, dass ein Programm die Informationen einlesen soll, wurde entschieden, dass der Bericht in Form eines JSONs gespeichert wird. Ein Befund im JSON ist wie folgt definiert:

- Sicherheitsrisiko (Rank)
- Letzte Projektklasse (LastProjectClassName)
- Letzte Projektmethode (LastProjectMethodName)
- Sicherheitskritische Klasse (VulnerabilityClass)
- Sicherheitskritische Methode (VulnerabilityMethod)
- Ausgenutzte Klasse (ExploitableClass)
- Ausgenutze Methode (ExploitableMethod)
- Vorherige Abweichungen (DfferenceMethods)
- Gefährliche Parameter (DangerousParameters)

Das Sicherheitsrisiko ist die berechnete Gefahr aus den im Unterkapitel 4.2.3 beschriebenen Verfahren für die Analyse von Methoden und Parameterwerten. Die letzte Projektklasse und -methode sind zusätzlich vom Nutzen, falls eine Schwachstelle innerhalb einer Bibliothek auftritt. In diesem Fall ist es sinnvoll zu wissen wie die entsprechende Bibliothek aufgerufen wurde, umso die Sicherheitslücke besser beheben zu können. In dem JSON ist zum einen die schwachstellen behaftete Methode, die es ermöglicht von der Programmausführung abzuweichen, als auch die Methode die durch die Differenz ausgenutzt wird um den Exploit durchzuführen. Wie in dem Paper *Vulnerability Recognition by Execution Trace Differentiation* [19] von Fabien Patrick Viertel et al. handelt es sich bei der letzten Methode in der regulären Programmausführung um die schwachstellen behaftete Methode, da es ab diesem Methodenaufruf eine Differenz gibt. Die Herangehensweise ist auch auf den Runtime-Security-Monitor zu übertragen, da die letzte Methode in dem statischen Modell, die letzte Methode war die von dem Entwickler vorgesehen wurde.

Zusätzlich enthält der Bericht noch gegebenenfalls Methoden bei denen schon vorherige Differenzen erkannt wurden, die aber nicht unmittelbar etwas mit der eigentlichen Differenz zutun haben, jedoch aber trotzdem mit dem Exploit in Verbindung stehen können. Die weiteren Abweichungen können so hilfreich sein um genauere Informationen zu erhalten, wie der Exploit

zustande gekommen ist. Wenn die Parameteranalyse aktiviert ist, werden auch noch gefundene Treffer mit in den Befund aufgenommen.

Kapitel 5

Implementierung

In dem Kapitel Implementierung wird erklärt wie der Runtime Security Monitor ausgehend vom Konzept umgesetzt wurde. Zuerst werden auf die grundsätzlichen Implementierungsentscheidungen eingegangen. In den Unterkapiteln wird dann näher auf die Implementierungen der einzelnen Phasen eingegangen.

5.1 Architektur

Der Runtime Security Monitor besteht aus zwei Projekten. Zum einem gibt es die Hauptanwendung die für die Modellerzeugung, heuristische Erkennung und der Berichterstellung zuständig ist. Des Weiteren gibt es noch den Java Agenten der die Klassen instrumentiert und so die aufgerufenen Methoden vom überwachenden Programm aufzeichnet, siehe Abbildung 5.1.

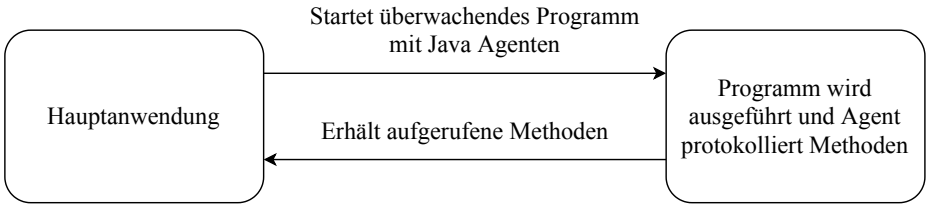

Abbildung 5.1: Architektur

Zum einen hat das den Vorteil, dass die Hauptanwendung und der Java Agent unabhängig von einander funktionieren und eine der Anwendungen ohne großen Aufwand ausgetauscht werden kann. Ein weiterer wichtiger Grund für das Aufteilen in zwei Anwendungen ist, dass der Agent mit einer anderen Java Version ausgeführt werden kann und die Hauptanwendung immer auf einer neuen Java Version laufen kann. So ist es Möglich, dass in der Hauptanwendung Sprachfunktionen aus Java 8 genutzt werden, gleichzeitig aber können auch Java-Programme mit einer niedrigeren Version überwacht

werden. Die Agenten wurden in der Java Version 1.5 eingeführt und ab der Version 1.5 und neuer ist es daher möglich Programme zu überwachen.

Es wurde sich in der Implementierung dazu entschieden, dass zum Start der Hauptanwendung alle wichtigen Informationen zur Programmüberwachung in Form einer Konfigurationsdatei übergeben werden. Das hat den Vorteil das keine grafische Oberfläche benötigt wird. So gibt es für ein Programm das überwacht werden soll eine entsprechende Konfigurationsdatei. In der Konfigurationsdatei können Einstellungen wie die verwendete Java-Version, Datenbank Zugangsdaten, Dateinamen, Optionen wie das Parsen von Parameterwerten und je nach Anwendung Black- und Whitelisten für Packages und Jar-Dateien gesetzt werden.

5.2 Domänenmodell

Für die Entwicklung wurde zuerst ein Domänenmodell erstellt, siehe Abbildung 5.2. Das Domänenmodell ist in die drei Bereiche Modellerzeugung, Heuristische Erkennung und Tracing aufgeteilt. Aus den abgebildeten Entitäten wurden dann die entsprechenden Klassen erstellt. Dabei wurde aus Gründen der Übersichtlichkeit auf die Methoden verzichtet. Durch die Pfeile werden die Beziehungen der einzelnen Klassen verdeutlicht. So besitzt die Klasse *CallGraph* zum Beispiel das Klassenattribute *classes*, dass auf mehrere Objekte von der Entität *Clazz* referenziert.

Modellerzeugung
Als Ausgangspunkt in dem Bereich der Modellerzeugung ist die Klasse CallGraph. Ein CallGraph besitzt als Attribut die Menge der Klassen, die sich in dem CallGraph befinden. Die Entität Clazz besitzt den Namen, sowie ihre entsprechenden Methoden. Eine Methode selbst wird mit dem Namen und der Methodensignatur identifiziert und besitzt mehrere Methodenaufrufe. Ein Methodenaufruf besitzt einen Type bei dem es sich um die jeweilige Instruktion aus dem Bytecode handelt, siehe Tabelle 2.1. Außerdem hat ein Methodenaufruf ein Klassenname, Methodenname und sowie die Signatur.

Tracing
In dem Package Tracing gibt es die Klasse TracingLog und TracingBuffer. Beide sind für einen speziellen Fall notwendig. Falls es sich bei dem Speicherungsformat um eine Datei handelt wird die TraceingLog Entität verwendet. Wenn dahingegen in echt Zeit geprüft werden soll, so wird der TracingBuffer verwendet. Ein TracingCall hat sehr viele Attribute, in dem Kapitel 5.4 wird näher auf die Entität eingegangen.

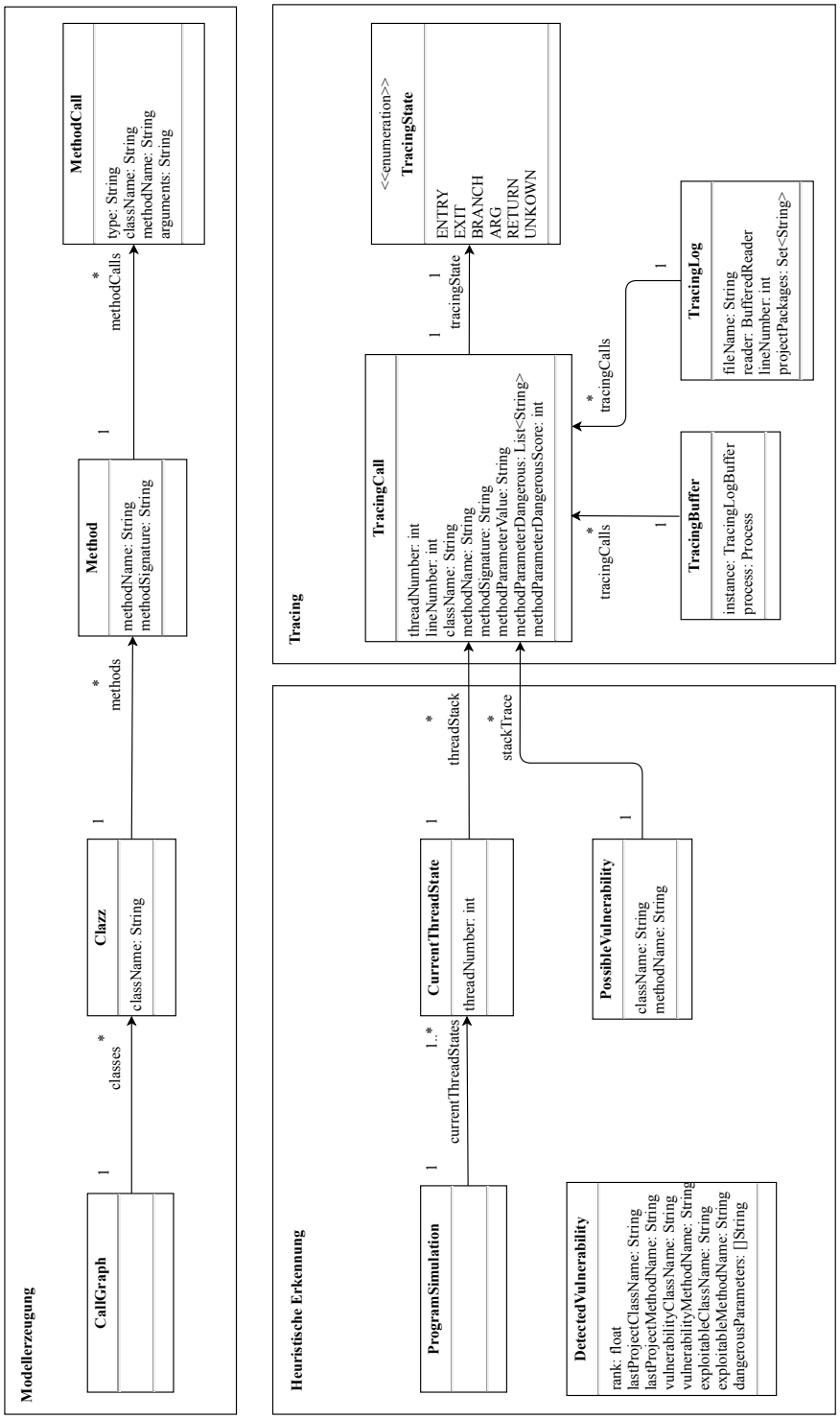

Abbildung 5.2: Domänenmodell

Heuristische Erkennung
Die Heuristische Erkennung besteht hauptsächlich aus vier Entitäten. Die Klasse ProgramSimulation hat die Aufgabe die aktuellen Threads zu verwalten. Aus diesem Grund hat die Klasse mindestens den Hauptthread, sowie noch weitere Threads, die während der Programmausführung erzeugt wurden. Sobald eine Abweichung zwischen dem Call Graph und einem CurrentThread erkannt wurde, wird ein Objekt von der Klasse PossibleVulnerability erzeugt. Eine PossibleVulnerability besitzt als Attribute den Klassennamen und Methodenname der vom CallGraph abweicht, sowie zusätzlich noch den Stacktrace. Der Stacktrace wird deshalb benötigt, um darzustellen wie die Abweichung entstanden ist. Die Klasse DetectedVulnerability wird für den Bericht benötigt. Wie im Kapitel 4.2.3 beschrieben wird mittels einer Bewertung eingestuft wie kritisch die erkannte Schwachstelle ist. Des Weiteren sind in der Klasse noch die wichtigen Klassen- und Methodennamen, die bereits im Kapitel 4.2.4 erklärt wurden. Falls die Option für die Auswertung von Parametern aktiviert ist und Parameter mit verdächtigen Inhalt erkannt wurden, befinden sich diese zusätzlich in dem Attribut dangerousParameters.

5.3 Modellerzeugung

Über die Konfigurationsdatei wird zum Programmstart die zu überwachende JAR-Datei übergeben. Um aus dem Java Archive einen statischen Aufrufgraphen zu erstellen wurde dieses nicht komplett neu entworfen, sondern die Java-Callgraph Bibliothek [20] von Georgios Gousios verwendet. Diese Bibliothek wiederum selbst nutzt die Byte Code Engineering Library (Apache Commons BCEL™) von der Apache Software Foundation. Mithilfe der Byte Code Engineering Library kann Java Bytecode effizient analysiert werden. In dem Java Archive werden so alle Bytecode Dateien, die mit der Dateiendung ".class" zu erkennen sind, eingelesen. Für die Umsetzung wurde von der Java-Callgraph Bibliothek das Visitor-Pattern verwendet. Zum einen gibt es ein Visitor für die einzelnen Klassen. Dieser Visitor erstellt bei jeder besuchten Methode, innerhalb einer Klasse, einen neuen weiteren Visitor der jede einzelne Methode behandelt. Innerhalb dieses MethodVisitors werden die einzelnen Bytecode Instruktionen besucht. Ausgehend davon werden diese im entwickelten Modell vom Runtime Security Monitor hinzugefügt. Dabei ist während der Entwicklung aufgefallen, dass Typumwandlungen, wie das explizite Casten, nicht berücksichtigt wurden. Aus diesem Grund wurde der Methoden Visitor erweitert, sodass auch Methodenaufrufe bei denen sich die konkrete Klasse ändert, mit in den Aufrufgraphen aufgenommen werden.

Ein Java Archive kann neben den Bytecode Dateien zusätzlich noch weitere Jar-Dateien beinhalten. Das wurde von der Java-Callgraph Bibliothek nicht vorgesehen, wodurch die Bibliothek erweitert werden musste.

5.4 Tracing

Wie in Kapitel 4.2.2 Tracing aus dem Konzept erläutert, soll eine Anwendung mittels der Java Instrumentation Schnittstelle überwacht werden. Durch den Call Graph lassen sich alle Klassen ausgeben, die in der zu überwachenden Anwendung verwendet werden. Diese Klassen müssen instrumentiert werden, damit protokolliert werden kann welche Methoden aufgerufen werden. Für das Tracing wurde das Tool Intrace [21] verwendet. Intrace selbst setzt auf das Java Bytecode Manipulations- und Analyse-Framework *ASM*[1]. Gerade im Bezug auf Speicherung der Traces gab es zunächst drei Möglichkeiten:

1. Speicherung in das Dateisystem
2. Speicherung in eine Datenbank
3. Übergabe vom Kindprozess mithilfe vom Scanner an die Hauptanwendung

Die erste Variante das Schreiben von Traces in das Dateisystem wird standardmäßig von Intrace unterstützt. Für das Speichern von Traces, um diese dann später für das Erweitern vom Call-Graph zu nutzen, hat sich die Option als besonders gut erwiesen, da die Traces nicht sofort eingelesen werden müssen und die einzelnen Aufrufe schnell aus dem Agenten schreibt. Das Speichern in eine Datenbank wurde nicht standardmäßig von Intrace unterstützt. Dafür wurde eine MySQL Verbindung in den Agenten integriert und die *OutputSettings*, in der die Varianten zur Auswahl stehen, entsprechend erweitert. Jedoch hat sich herausgestellt, dass der Agent und die damit verbundene Performance in der Praxis schlechter ist, als wie es bei den anderen zwei Möglichkeiten der Fall ist. Deshalb wurde sich dafür entschieden das die Inhalte über die Standardausgabe ausgegeben werden und diese dann von der Hauptanwendung eingelesen werden. Das Verfahren kann deshalb durchgeführt werden, da der Agent als Kindprozess gestartet wurde und so der Datenempfang mithilfe eines Scanners ermöglicht wird.

Darüber hinaus unterstützt Intrace von Haus aus keine Signaturen. Ein Methodenaufruf besteht nur aus dem Klassen- und Methodennamen. In Java gibt es zu dem, aber die Möglichkeit Methoden zu überladen. Ohne die Signatur würde es nicht möglich sein eine konkrete Aussage darüber zu treffen, welche Methode der überladenen Methode aufgerufen wurde. Um das Problem zu beheben wurde der *InstrumentedClassWriter* überarbeitet. Durch die Überarbeitung wird eine weitere LDC-Instruktion in den Bytecode hinzugefügt, der die fehlende Signatur aus den Konstantenpool auf den Stack lädt. Anschließend wird der Methodenaufruf erzeugt, der die zusätzliche Methodensignatur enthält. Bei dem Methodenaufruf handelt es

[1] `https://asm.ow2.io/`

sich um eine zentrale Hilfsmethode von dem Agenten der den protokollierten Methodenaufruf dann weiter verteilen kann.

Jeder protokollierte Aufruf (TracingCall) besteht aus der Thread-Nummer, den Klassen- und Methodennamen sowie einen TracingState. Der TracingState defeniert ob es sich um einen Methodeneintritt oder -austritt handelt. Zusätzlich kann ein TracingCall auch vom TracingState Argument sein. Wenn die Parameteranalyse in der Konfigurationsdatei aktiviert wurde, folgen nach jedem Methodeneintritt, immer so viele TracingCalls vom TracingState Argument, wie die Methode Parameter besitzt.

5.5 Heuristische Erkennung

Für die heuristische Erkennung von Schwachstellen wurde sich entschieden die protokollierten Methodenaufrufe die zur Laufzeit aufgetreten sind mithilfe den Aufrufen aus dem statischen Call Graph abzugleichen. Dafür wurde eine Technik entworfen, die die Daten von dem Tracing entgegen nimmt und während der Laufzeit das Modell abgleicht.

Um das Verfahren zu realisieren hat jeder Thread aus der zu überwachenden Anwendung in dem Runtime-Security-Monitor einen eigenen Stack in dem sich die Methodenaufrufe befinden. So kann der aktuelle Zustand des laufenden Programms abgebildet werden, siehe Abbildung 5.3. Ruft eine Methode, wie der Konstruktor von der Klasse *Product*, eine andere Methode auf, so muss dieser Aufruf von der heuristischen Erkennung auf Validität geprüft werden, ob es sich um einen gültigen Aufruf handelt.

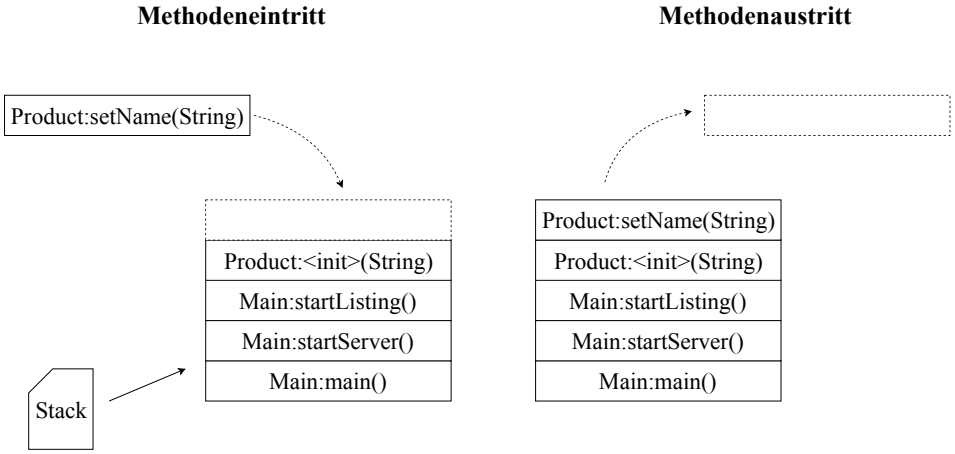

Abbildung 5.3: Stack: Methodeneintriit und Methodenaustritt

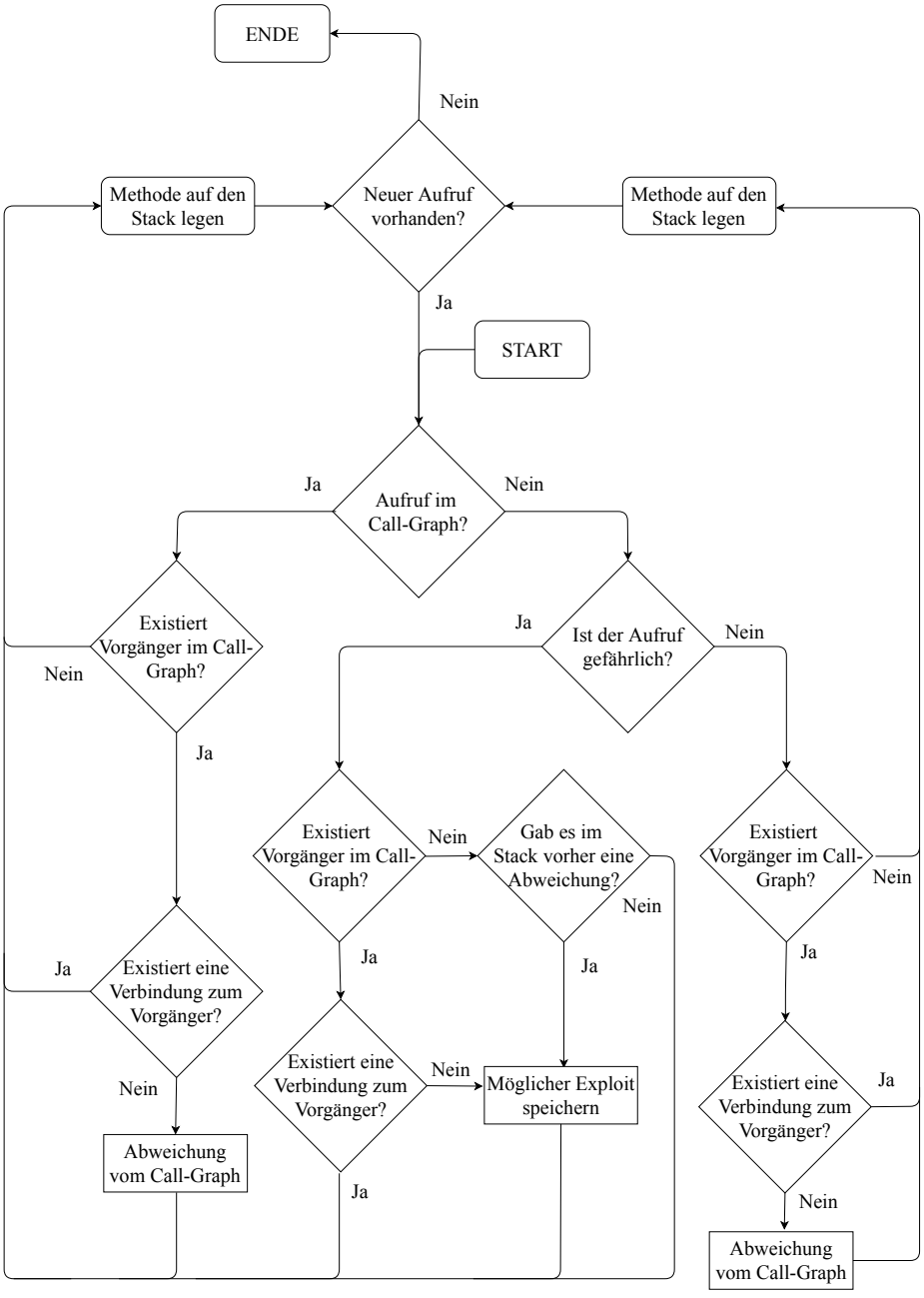

Abbildung 5.4: Programmablauf für Methodenaufruf

Für den Methodeneintritt wurde ein Programmablaufplan 5.4 erstellt, der die Ausführung der heuristischen Erkennung beschreibt. Nach dem Start wird zuerst geprüft, ob die aufgerufene Methode in dem Call-Graph enthalten ist. Falls ja handelt es sich um eine Methode die sich innerhalb des Projekts oder einer Bibliothek befindet. Im nächsten Schritt wird geprüft, ob die

Methode die bereits auf dem Stack liegt in dem Call-Graph enthalten ist. Falls dieses nicht der Fall ist so wurde die Methode aus der JRE aufgerufen. Befindet sich die auf dem Stack liegende Methode ebenfalls im Call-Graph so kann geprüft werden, ob die Methode eine Kantenbeziehung zu dem aktuellen Aufruf besitzt. Ist dieses der Fall so ist alles in Ordnung und die Methode kann auf dem Stack gelegt werden und es kann mit der nächsten Methode fortgesetzt werden. Ist allerdings keine Verbindung zwischen den beiden Methoden vorhanden, so handelt es sich um eine Abweichung, da in der Entwicklung nicht vorgesehen wurde, dass die auf dem Stack liegende Methode die andere Methode aufruft. Ist hingegen die zu prüfende Methode nicht im Call-Graph vorhanden, so kann es sich um eine gefährliche Methode handeln, die sich in dem JRE befindet. Falls die Methode gefährlich ist, wird auch hier wieder geprüft, ob die auf dem Stack liegende Methode im Call-Graph vorhanden ist. Falls ja kann geprüft werden ob die Methode auf dem Stack die aktuell aufzurufende Methode aufrufen darf. Falls ja kann wieder normal fortgesetzt werden. Gibt es allerdings keine Verbindung so kann es sich wahrscheinlich um ein Code-Exploit handeln, da nach Modell die Methode nicht aufgerufen werden kann. Ist sowohl der aktuell zu prüfende Methodenaufruf, sowie die Methode auf dem Stack nicht im Call-Graph, so handelt es sich um einen Aufruf der im JRE eine andere JRE Methode aufruft. Wurden vorher schon Abweichungen gefunden, so kann es sich in diesem Fall auch um einen Exploit handeln. Falls nicht kann mit dem nächsten Aufruf fortgesetzt werden. Handelt es sich bei der zu prüfenden Methode um keine gefährliche Methode, so wird wie schon in den anderen Verzweigungen geprüft ob es eine Verbindung gibt. Falls nicht so wird diese Abweichung gespeichert und es kann mit der nächsten Methode fortgesetzt werden.

5.6 Bericht und Speicherung

Die potenziellen Schwachstellen werden zunächst in einer Datenbank gespeichert. Sobald das zu überwachende Programm beendet ist, wird ein Bericht erstellt. Dieser Bericht wird als JSON gespeichert und ist wie nach dem Konzept beschrieben implementiert worden.

```
1  [{
2    "rank": 4.5,
3    "lastProjectClassName":"io.meterian.samples.jackson.Main",
4    "lastProjectMethodName": "deserialize",
5    "vulnerabilityClass": "org.apache.xalan.xsltc.trax.
         TemplatesImpl",
6    "vulnerabilityMethod": "getTransletInstance",
7    "exploitableClass": "java.lang.Runtime",
8    "exploitableMethod": "exec",
9    "differenceMethods": [
10     ["com.fasterxml.jackson.databind.deser.impl.
           SetterlessProperty", "deserializeAndSet"],
11     ["org.apache.xalan.xsltc.trax.TemplatesImpl", "
           getTransletInstance"]
12   ],
13   "dangerousParameters": ["C:/Windows/System32/calc.exe"]
14 }]
```

JSON 5.1: Befund eines Code-Exploits

5.7 Optimierung

Gerade am Ende der Entwicklungsphase wurden gezielt Optimierungen durchgeführt, die noch mal versucht haben die Anwendung zu beschleunigen. Dabei wurden vorallem Datenstrukturen wie Listen durch HashMaps ersetzt. Die HashMaps ermöglichen eine deutlich schnellere Zugriffszeit, wenn zum Beispiel in dem Call-Graph nach einer Klasse gesucht wird, muss nicht die ganze Liste überprüft werden, was sonst einer Zugriffzeit von *O(n)* entspricht. Wenn der Name zur suchenden Klasse vorhanden ist, so wird bei einer HashMap im bestfall nur eine konstante Zugriffszeit *O(1)* benötigt. Des Weiteren wurde versucht die Datenbankenanfragen zu minimieren, in dem Statements erst gesammelt wurden und dann nach einer gewissen Zeit gebündelt an den Datenbankenserver versendet wurden.

Kapitel 6

Evaluation

Ein wichtiger Bestandteil dieser Arbeit ist es das entwickelte Programm zu evaluieren. Es wurde sich dafür entschieden bereits bekannte Schwachstellen mit dem Programm zu erkennen, umso Rückschlüsse auf die beiden Faktoren Genauigkeit und Laufzeiten zu ziehen, die in den Kapiteln 6.1 und 6.2 näher erläutert werden. Die Schwachstellen wurden bereits veröffentlicht und sind in dem Common Vulnerabilities and Exposures (CVE) Verzeichnis eingetragen. Um die Schwachstellen auszunutzen, werden Exploits benötigt. Anzumerken ist das es sich bei der Evaluierung auf vier Exploits beschränkt wurde, da für die Evaluation einerseits der Exploit-Programmcode benötigt wird, um die Schwachstelle auszunutzen, andererseits aber auch ein lauffähiges Programm zur Verfügung stehen muss. Die Recherche hat sich als sehr zeitaufwendig erwiesen. Ein Grund dafür war das sich die Bachelorarbeit auf Programme fokussiert hat, die als JAR-Datei gebaut wurden und sich damit mittels einer Main Methode ausführen lassen. Web Application Archive, kurz WAR-Datei, können leider nicht ohne Weiteres dem Programm übergeben werden, da ihnen die Main Methode fehlt, die aber zwingend erforderlich ist damit ein Einstiegspunkt ermöglicht wird. Eine WAR-Datei besitzt eine web.xml, in der einzelne Servlets definiert sind, die dann Anfragen entgegennehmen können. Deshalb muss das schwachstellen behaftete Programm so umgebaut werden, sodass es eine Main Methode besitzt und sich damit dann ausführen lässt. Dieses wurde für ein Exploit mittels eines Embedded Servers umgesetzt.

Exploit 1: CVE-2015-6420
In der Apache Commons Collections Version 3.2.1 ist es möglich durch den InvokerTransformer der sich in der Bibliothek befindet, eine Code-Execution durchzuführen [22]. Wenn eine Anwendung mit der Bibliothek einen ObjectInputStream erhält und die Anwendung über die Methode readObject versucht das Objekt zu erstellen, kann es sich bei dem Objekt um eine TranformedMap handeln. Die TransformedMap besitzt mehrere

InvokerTransformer und führt dann über die Methode tranform den Code-Exploit aus. Da sich die Bibliothek auch in stark verbreiteten Anwendungen wie JBoss befindet, waren sehr viele Programme unsicher. Eine Anwendung, die den Exploit beinhaltet war in dem Institut vorhanden und konnte für die Evaluation genutzt werden.

Exploit 2: CVE-2017–5638
Bei dem Apache Structs 2 handelt es sich um ein Webframework, dass für das erstellen von Webanwendnungen und Webseiten genutzt wird. In den Versionen 2.3.1 – 2.3.31 und 2.5 bis 2.5.10 von Apache Structs 2 kann über den Content-Type Schadcode in das System eingeschleust werden, wenn es sich bei einer Client-Anfrage um den Content-Type "multipart/form-data" handelt [23]. Der Multipart-Parser Jakarta in Apache Structs 2 führt dann den zusätzlichen Inhalt in dem Content-Type aus und ermöglicht so einen Code-Exploit. Apache bietet eine Showcase App an, die für die Evaluation genutzt wurde. Außerdem wurde der entsprechende Exploit von der Exploit Database[1] verwendet. Da Apache Structs 2 ein Webframework ist, musste die Showcase App mittels eines Embedded Server gestartet werden.

Exploit 3: CVE-2017-7525
In dem Jackson-Databind Version 2.8.8 gibt es die Möglichkeit das über Deserialisierung böswilliger Programmcode ausgeführt werden kann. Dabei wird in dem ObjectMapper mit der Methode readValue von Jackson-Databind versucht ein Objekt zu erzeugen, die davon ausgehend Schadcode einschleust und zum Ausführen bringt [24]. Eine Anwendung und Exploit existiert auf GitHub[2].

Exploit 4: CVE-2017-8046
Über den letzten Exploit in der Evaluation kann durch eine Schwachstelle in dem Spring Data REST ebenfalls Code-Execution durchgeführt werden. Das Modul Spring Data REST ist eine Bibliothek die in Spring Boot Applikationen verwendet werden kann. Wird das Modul verwendet, so kann eine Serveranfrage mit einem speziell zusammengebautem JSON an das Spring Boot Projekt gesendet werden, umso Programmcode auszuführen [25]. Eine bereits lauffähige Spring Boot Anwendung, die das Spring Data REST Modul enthält, sowie ein Code-Exploit existiert auf GitHub[3].

[1] Exploit für CVE-2017–5638: `https://www.exploit-db.com/exploits/41570`

[2] Anwendung und Exploit für CVE-2017-7525: `https://github.com/bbossola/vulnerability-java-samples`

[3] Anwendung und Exploit für CVE-2017-8046: `https://github.com/m3ssap0/spring-break_cve-2017-8046`

6.1 Genauigkeit

In diesem Kapitel wird darauf eingegangen wie präzise das Programm ein Code-Exploit erkennen kann, und was für Einschränkungen gelten. Da das Verfahren von dem Runtime Security Monitor darauf beruht am Anfang ein statisches Modell zu erstellen, würde jede Abweichung in der Laufzeit als potenzielle gefährliche Differenz zu werten sein, weil ein Code-Exploit die Abweichung verursacht haben könnte. Wie schon in dem Kapitel Heuristische Erkennung von Schwachstellen 4.2.3 beschrieben wird versucht mit verschiedenen Ansätzen eine Kategorisierung durchzuführen. Um das Programm zu evaluieren, wurde sich entschieden dieses mittels den Maßen Recall und Precision durchzuführen, die aus dem Bereich des Information Retrievals stammen. Die Autoren Christopher D. Manning et al. in dem Buch *Introduction to Information Retrieval* [26] das Verfahren beschrieben. Die konkrete Wahrheitsmatrix in Tabelle 6.1, klassifiziert einen möglichen Code-Exploit in 4 unterschiedliche Klassen.

	Relevant (Code-Exploit)	**Nicht relevant (Kein Code-Exploit)**
Gefunden	True Positives (TP)	False Positives (FP)
Nicht gefunden	False Negatives (FN)	True Negatives (TN)

Tabelle 6.1: Wahrheitsmatrix

Dabei ist die True Positives(TP) Klasse die, bei der das Programm den Code-Exploit gefunden hat und es sich auch tatsächlich um ein Code-Exploit handelt. False Positives(FP) sind fälschlicherweise als Code-Exploit erkannt, die jedoch keinen Exploit darstellen. False Negatives sind Code-Exploits die nicht von dem Programm erkannt wurden. Die Klasse True Negatives(TN) spielt in der Evaluation keine Rolle, da sie für die Berechnung von Recall und Precision nicht wichtig ist, weil die Klasse die Ergebnisse beinhaltet die nicht relevant und nicht gefunden wurden. Precision(P) und Recall (R) sind nach den Autoren wie folgt definiert:

$$\text{Recall (R)} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\#(\text{Gefundene, Relevate Ergebnisse})}{\#(\text{Relevante Ergebnisse})}$$

$$\text{Precision (P)} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\#(\text{Erhaltene, Relevate Ergebnisse})}{\#(\text{Erhaltene Ergebnisse})}$$

Dabei handelt es sich bei Recall um den prozentualen Wert, wie viele Code-Exploits tatsächlich von dem Programm erkannt wurden. Ein hoher Wert wäre strebenswert, da so garantiert wird das auch alle Code-Exploits

gefunden wurden. Bei Precision wird eine Aussage darüber getroffen, ob es sich bei dem Ergebniss auch tatsächlich um einen Exploit handelt. Werden durch die vorgestellten Ansätze Abweichungen vom statischen Modell als Code-Exploits erkannt, bei den Abweichungen es sich jedoch um keinen Code-Exploits handelt, so sinkt der Wert der Precision.

Um aus den beiden Maßen ein Gesamtmaß zu berechnen, wird ein harmonisches Mittel gebildet, als F1 genannt. Dabei sind die beiden Maßen gleich stark gewichtet:

$$\mathrm{F1} = \frac{2\mathrm{PR}}{\mathrm{P} + \mathrm{R}}$$

Für die Evaluation wurden die verschiedenen Ansätze zur Kategorisierung evaluiert, siehe Tabelle 6.2. Um das Verfahren konkret an einem Beispiel zu verdeutlichen wurde die erste Zeile der Tabelle nachfolgend erklärt. Es wurden alle vier Anwendungen gestartet und die Exploits durchgeführt. Dabei wurde für den Recall sich auf die vier bekannten Exploits beschränkt. Da alle Exploits vom Runtime Security Monitor erkannt wurden, liegt der Recall bei 100%.

$$\text{Recall (R)} = \frac{4}{4+0} = 100\%$$

Die Präzision berechnet sich aus den vier erkannten Exploits sowie zusätzlich zwei Ergebnissen, bei denen das Programm ein Code-Exploit vermutet hat. Es handelt sich hierbei aber um keinen Exploit.

$$\text{Precision (P)} = \frac{4}{4+2} = 66\%$$

Da beide Maßen gegenseitig im Verhältnis stehen, wurde das harmonische Mittel gebildet und das F1 beträgt 80%.

$$\mathrm{F1} = \frac{2\mathrm{PR}}{\mathrm{P} + \mathrm{R}} = 80\%$$

Kategorisierungsverfahren	Recall	Precision	F1
Kritische Methoden von OWASP	100 %	66 %	80 %
Kritische Methoden von OWASP & mit dynamischen Call-Graph	100 %	100 %	100 %
Kritische Methoden von Entroine	100 %	1 %	1 %
Kritische Methoden von Entroine & mit Parameterwert Analyse Threshold 3.0	100 %	80 %	89 %

Tabelle 6.2: Evaluierung der Kategorisierungen

6.1.1 Interpretation

Allgemein lässt sich sagen, dass der Recall von 100% sehr gut ist. Es konnten so immer alle Code-Exploits erkannt werden. Jedoch gibt es mit den unterschiedlichen Kategorisierungsverfahren bei der Precision große Unterschiede. Die Precision bei der Analyse von den kritischen Methoden von OWASP liegt bei 66 %. Bei der Analyse wurde ein Code-Exploit bei Spring Data REST und Apache Structs 2 falsch erkannt, weil die beiden Aufrufe von einem URLClassLoader nicht im statischen Call-Graph vorhanden waren. Da der URLClassLoader ausgenutzt werden könnte um eine beliebige Klasse zu laden, ist das Gefahrenpotenzial sehr hoch. Wenn man die Anwendungen aber aufzeichnet und die dabei zur Laufzeit entstandenen Abweichungen zum Call-Graph hinzufügt, wird nicht mehr von einem Code-Exploit ausgegangen. Die Precision von der Heuristischen Erkennung mithilfe der kritischen Methoden von Entroine sind sehr gering und liegt bei 1 %. Grund dafür ist das bei Entroine zusätzlich noch mehr gefährliche Methoden geprüft werden und die Abweichungen vom Call-Graph ebenfalls mit in die Kategorie Code-Exploit entschieden werden. Das F1 Maß fällt mit 80 % und 100 % gut aus, jedoch ist das F1 Maß von Entroine sehr schlecht zu interpretieren, da die Precision nicht gut ist. Verwendet man jedoch zusätzlich noch die Parameteranalyse so kann eine bessere Klassifizierung getroffen werden, ob es sich tatsächlich um einen Code-Exploit handelt. Die Precision steigert sich auf 80 % und aufgrund der besseren Precision ist auch das F1 Maß mit 89% entsprechend gut.

6.2 Laufzeiten

Für die Messungen in dem Kapitel wurde folgende Hardware verwendet:

Betriebsystem:	MacOS Version 10.14.4
Prozessor:	Intel Core i7-4870HQ, Takt 2.5 GHz
Arbeitsspeicher:	16GB DDR3
Festplatte:	Apple SM0256G

Tabelle 6.3: Verwendete Hardware

6.2.1 Agenten Laufzeit

Gerade die Laufzeiten der Programme, die überwacht werden, spielen einen hohen Stellenwert. Während der Konzept- und Entwicklungsphase ist aufgefallen, dass die Performance je nach Komplexität der Anwendung unterschiedlich stark ins Gewicht fällt. Aus diesem Grund wurden die zwei Filteroptionen, die in den vorherigen Kapiteln als die Listen für die gefährlichen Methoden definiert wurden, miteinander verglichen, umso Rückschlüsse

auf entsprechende Laufzeiten zu bekommen. Dabei werden bei den Listen nur die Klassen von der Java Runtime Environment instrumentiert, die bekannt für Exploits sind. Da der Agent eine wichtige Rolle in Bezug auf die Performance spielt, wurde dieser gezielt evaluiert.

CVE-ID	Ohne Agent	OWASP	Entroine	Alle Klassen
CVE-2015-6420	488ms	1085ms	2615ms	5492ms
CVE-2017–5638	5,856s	10,944s	131,680s	688,352s
CVE-2017-7525	1,184s	8,166s	23,379s	175,835s
CVE-2017-8046	16,355s	212,069s	939,025s	3611,228s
CVE-2017-8046 (Unter Vermeidung von Overhead)		18,545s	150,890s	1024,926s

Tabelle 6.4: Laufzeiten

In der ersten Spalte "Ohne Agent" wurde der Exploit gemessen, dabei wurden aber noch keine Methoden protokolliert, dieser Wert soll einen Referenzwert zur Echtzeit bilden. In OWASP wurden alle Projektklassen und die JRE-Klassen instrumentiert, die von dem OWASP, als anfällige Klassen für Code Execution stehen. Entroine besitzt zusätzlich zu der OWASP Liste noch weitere Methoden, bei denen es sich um Exploitable Methods handelt. Alle Klassen bezeichnet die Ausführungszeit, wenn alle Methoden die direkt mit dem Call-Graph in Verbindung stehen instrumentiert werden.

6.2.2 Instrumentierung von Methoden

Die Anzahl der Methoden und wie viele davon zur Laufzeit wirklich instrumentiert werden, wurde mit dem Tool OWASP Code Pulse [27] berechnet, siehe Tabelle 6.5. Code Pulse wird für das Erkennen von Code Coverage zur Laufzeit verwendet. Mit dem Tool lässt sich so schnell erkennen welche Methoden oder ganze Packages aufgerufen wurden.

Das Starten der Anwendung von Spring Boot hat mit Code Pulse 108 Sekunden gedauert. Der Runtime Security Monitor braucht dafür 53 Sekunden. Bei diesen Zeiten wird aber noch keine Bibliothek (JAR-Datei in JAR-Datei) dazu geladen, es handelt sich nur um den Spring Boot Loader und den Embedded Server. Aus diesem Grund sind in der letzten Spalte von der Tabelle 6.5 lediglich 415 Methoden. Zählt man alle Methoden in der Anwendung von Spring Boot und den zusätzlichen Bibliotheken zusammen so handelt es sich um etwa 148.000 Methoden.

CVE-ID	Schwachstelle in:	Methoden	Instrumentiert
CVE-2015-6420	Apache Commons Collections	4.000	30
CVE-2017–5638	Apache Structs 2	6.100	1.037
CVE-2017-7525	Jackson Databind	39.000	2.340
CVE-2017-8046	Spring Data REST	415	251

Tabelle 6.5: Methoden

6.2.3 Interpretation

Zusammenfassend lässt sich feststellen, dass sobald eine zu analysierende Anwendung sehr viele Methoden besitzt, zusätzlich in vorm von Bibliotheken oder auch JRE-Methoden, ist es nicht mehr möglich, alle Klassen zu instrumentieren. Hier muss dann die Auswahl mittels bestimmter Listen eingeschränkt werden, da sonst die Ausführungszeit unverhältnismäßig lange dauert. Besonders hervorzuheben ist das bei einer eher kleinen Anwendung, wie es bei CVE-2015-6420 der Fall ist, der Anstieg eher proportional im Verhältnis zu den Einstellungen steht, wo hingegen der Anstieg bei CVE-2017–5638 deutlich stärker ins Gewicht fällt. Das ist vor allem darauf zurückzuführen, da die zusätzlichen Klassen aus dem JRE in einer großen Anwendung öfters aufgerufen werden, weil die Anwendung stärker auf diesen Klassen aufbauen. Bei CVE-2017-8046 wird von Spring Boot unter anderem sehr oft die Methode “Swap“ aufgerufen, und da bei jedem Aufruf die Methode geprüft werden muss, verursacht die Methode einen deutlichen Overhead und erhöht die Laufzeit erheblich. In der Zeit wurden mit dem Filter 1 alleine knapp 24.000.000 Methodenaufrufe aufgezeichnet. Deshalb wurden die Methoden entfernt die für einen deutlichen Overhead gesorgt haben, wodurch die Performance verbessert werden konnte, siehe Tabelle 6.4 Laufzeiten

Kapitel 7

Zusammenfassung und Ausblick

In diesem Kapitel wird eine Zusammenfassung der vorliegenden Arbeit gegeben, sowie werden mögliche Risiken auf den Bezug der Validität der Ergebnisse aufgezeigt. In dem letzten Abschnitt Ausblick wird darauf eingegangen welche zukünftigen Forschungsarbeiten sich aus dieser Arbeit ergeben und welche Verbesserungen durchgeführt werden können.

7.1 Zusammenfassung

Die bereits in dem CVE Verzeichnis eingetragenen Schwachstellen konnten mithilfe von dem Runtime-Security-Monitor gefunden werden. Jedoch bietet ein statisches Modell, wie ein Call-Graph, alleine nicht die Möglichkeiten exakt mit einer dynamischen Programmausführung abgeglichen zu werden, um einen möglichen Exploit eindeutig identifizieren zu können. Hierfür wurden weitere Heuristiken wie das Analysieren von Parameterwerten zur Laufzeit verwendet. Da diese aber sehr subjektiv sind und das Verfahren nur so gut ist, wie die verwendeten Suchbegriffe, die sich in den Parameterwerten befinden, könnte ein Angreifer diese Analyse auch relativ einfach mit Verschleierungstechniken umgehen und hätte damit die Sicherungen relativ einfach umgangen. Da Java eine höhere Programmiersprache ist, können selbst auch schon kleine Anwendungen mit Bibliotheken durchaus schnell komplex werden, und es so schwierig machen, mit wenig vorhandenen Ressourcen alle Klassen zu analysieren und trotzdem schnell genug zu sein.

7.2 Risiken für die Validität

Das Finden von Exploits hängt stark davon ab, wie die Code-Exploits aufgebaut sind. Da gefährliche Methoden zur Identifizierung verwendet werden um einen Exploit zu erkennen, hängt dieses stark von der Auswahl

der gewählten Methoden ab. Deshalb wurde versucht sich bei der Auswahl an bereits existierenden Forschungsarbeiten zu bedienen.

Zudem sind bei der Parameterüberprüfung die einzelnen Suchbegriffe meist sehr subjektiv und hängen stark von der verwendeten Anwendung ab. Aus diesem Grund wurde auch versucht die einzelnen Begriffe nach der potenziellen Gefährlichkeit zu gewichten.

Auch ist die Anzahl der vier Code-Exploits möglicherweise nicht repräsentativ genug, da dieses nur eine kleine Testmenge darstellt. Es wurden daher auch Programme untersucht, bei denen der Call-Graph durch den massiven Einsatz von Reflection bei weitem nicht den optimal Fall entspricht.

7.3 Ausblick

Das Programm ist in der Lage Code-Exploits zu erkennen und dem Entwickler über mögliche Schwachstellen behaftete Methoden zu informieren. Jedoch hat sich auch in der Evaluation gezeigt, dass gerade durch den Einsatz von Reflection ein Code-Exploit nur unzureichend bestimmt werden kann, und die vorgestellten Verfahren zur Erkennung nur eingeschränkt objektiv sind und je nach Anwendungsfall stark variieren könnten. Ein möglicher Ansatz könnte sein, dass man mithilfe von einer künstlichen Intelligenz erkennt, ob es sich bei dem geänderten Verhalten um eine gutartige Reflection handelt. Zudem könnte versucht werden die Performance von dem Agenten noch weiter zu verbessern, sodass dieser auch mit großen Anwendungen und wenig Ressourcen gut funktioniert, damit dieser besser eingesetzt werden kann.

Da sich vor allem die Web Application Archive stark in dem strukturellen Aufbau zu einem Java Archive unterscheiden, kann es durchaus auch sinnvoll sein ein Verfahren zu entwickeln, das ein spezielles Modell erzeugt, um das Verhalten mit in Verbindung vom WebappClassLoader von Web Application Archive besser abzubilden.

Zu dem könnten weitere gefährliche Methodenlisten speziell für die Code-Exploits entworfen werden oder Parameteranalysen mit regulären Ausdrücken angelegt werden, die eine deutlich objektivere Darstellung bieten und genauer sind, um die Code-Exploits eindeutig zu erkennen.

Eine denkbare Option ist auch, dass die Anwendung die aktuell nur Java Programme überwachen kann, so zu erweitern, dass auch andere Programmiersprachen wie C++ zur Laufzeit auf mögliche Exploits überwacht werden können.

Anhang A

Anhang

A.1 DVD

Die Inhalte der beiliegenden DVD sind:

- Die schriftliche Arbeit in digitaler Form
- LATEX Dateien zu dieser erstellten Arbeit
- Den Runtime-Security-Monitor in ausführbarer Form
- Den Quellcode zu dem Programm
- Die Exploits sowie die Evaluationsergebnisse
- Internetquellen

Literaturverzeichnis

[1] Christian Ullenboom. *Java ist auch eine Insel - das umfassende Handbuch ; [aktuell zu Java 7].* Galileo Press, Bonn, 11. aufl. edition, 2012.

[2] Peter Schmitz. Cyberattacken auf deutsche Industrie nehmen stark zu. https://www.security-insider.de/cyberattacken-auf-deutsche-industrie-nehmen-stark-zu-a-766073/. letzer Zugriff: 15. Juli 2019.

[3] Philipp Herold. Daten in Gefahr: Angriffe auf IT-Systeme nehmen weiterhin zu. https://www.mein-datenschutzbeauftragter.de/blog/daten-in-gefahr-angriffe-auf-it-systeme-nehmen-weiterhin-zu/. letzer Zugriff: 15. Juli 2019.

[4] Stefan Beiersmann. Öffentliche verfügbare Exploits betreffen 90 Prozent aller produktiven SAP-Systeme. https://www.zdnet.de/88359229/oeffentliche-verfuegbare-exploits-betreffen-90-prozent-aller-produktiven-sap-systeme/, Mai 2019. letzer Zugriff: 15. Juli 2019.

[5] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification Java SE 8 Edition. https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf, March 2015. letzer Zugriff: 6. Juli 2019.

[6] Was ist ein Exploit? https://www.security-insider.de/was-ist-ein-exploit-a-618629/. letzter Zugriff 11. Juli 2019.

[7] OWASP. OWASP Top 10 - 2017, The Ten Most Critical Web Application Security Risks. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf, 2017. letzer Zugriff: 15. Juli 2019.

[8] Runtime Application Self-Protection (RASP). https://www.computerweekly.com/de/definition/Runtime-Application-Self-Protection-RASP. letzter Zugriff 11. Juli 2019.

[9] Was ist die heuristische Analyse? `https://www.kaspersky.de/resource-center/definitions/heuristic-analysis`. letzter Zugriff 11. Juli 2019.

[10] Funktionsweise der heuristischen Erkennung. `http://www.antivirenprogramm.net/grundlagen/funktionsweise-der-heuristischen-erkennung/`. letzter Zugriff 13. Juli 2019.

[11] William G. J. Halfond and Alessandro Orso. Combining static analysis and runtime monitoring to counter sql-injection attacks. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[12] Hajime Inoue and Stephanie Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the 2002 Workshop on New Security Paradigms*, NSPW '02, pages 52–60, New York, NY, USA, 2002. ACM.

[13] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. JavaParser: Visited. `https://leanpub.com/javaparservisited`, May 2019. letzer Zugriff: 6. Juli 2019.

[14] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, pages 107–112, New York, NY, USA, 2018. ACM.

[15] The AspectJtm Development Environment Guide. `https://www.eclipse.org/aspectj/doc/next/devguide/printable.html#ltw-specialcases`. letzter Zugriff 16. Juli 2019.

[16] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing java reflection. *ACM Trans. Softw. Eng. Methodol.*, 28(2):7:1–7:50, February 2019.

[17] OWASP. Searching for Code in J2EE/Java. `https://www.owasp.org/index.php/Searching_for_Code_in_J2EE/Java`. letzer Zugriff: 15. Juli 2019.

[18] G. Stergiopoulos, P. Petsanas, P. Katsaros, and D. Gritzalis. Automated exploit detection using path profiling: The disposition should matter, not the position. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 04, pages 100–111, July 2015.

[19] Fabien Patrick Viertel, Oliver Karras, and Kurt Schneider. Vulnerability recognition by execution trace differentiation. In *2017 ACM/IEEE*

International Symposium on Software Performance (SSP), Karlsruhe, 2017.

[20] Georgios Gousios. Java Call Graph Utilities. `https://github.com/gousiosg/java-callgraph`. letzter Zugriff 24. Juli 2019.

[21] Georgios Gousios. InTrace. `https://mchr3k.github.io/org.intrace`. letzter Zugriff 26. Juli 2019.

[22] Apache commons collections java library insecurely deserializes data. `https://www.kb.cert.org/vuls/id/576313/`, November 2015. letzer Zugriff: 17. Juli 2019.

[23] Exploiting apache struts2 cve-2017–5638 | lucideus research. `https://medium.com/@lucideus/exploiting-apache-struts2-cve-2017-5638-lucideus-research-83adb9490ede`, Oktober 2018. letzer Zugriff: 17. Juli 2019.

[24] Robert C. Seacord. Ncc group whitepaper jackson deserialization vulnerabilities. `https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2018/jackson_deserialization.pdf`, August 2018. letzer Zugriff: 17. Juli 2019.

[25] Man Yue Mo. Spring data rest exploit (cve-2017-8046): Finding a rce vulnerability with ql. `https://blog.semmle.com/spring-data-rest-CVE-2017-8046-ql/`, März 2018. letzer Zugriff: 17. Juli 2019.

[26] Christopher D. Manning, Hinrich Schütze, and Prabhakar Raghavan. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, 2008.

[27] OWASP Code Pulse Project. `https://www.owasp.org/index.php/OWASP_Code_Pulse_Project`. letzter Zugriff 26. Juli 2019.