WebTaint: Dynamic Taint Tracking for Java-based Web Applications

FREDRIK ADOLFSSON

Master in Computer Science
Date: June 7, 2018
Supervisor: Musard Balliu
Examiner: Mads Dam
Swedish title: WebTaint: Dynamic Taint Tracking för Java-baserade webbapplikationer
School of Computer Science and Communication

Todo: Information Disclosure -> Code Injection

Abstract

The internet is a source of information and it connects the world through a single platform. Many businesses have taken advantage of this to share information, communicate with customers, and create new business opportunities. However, this does not come without drawbacks. There are several potential attacks causing harm to web applications.

The thesis implemented a dynamic taint tracker, called WebTaint, to detect and prevent confidentiality and integrity vulnerabilities in Java-based web applications. We evaluated to what extent WebTaint can combat integrity vulnerabilities. The questions of what advantages and disadvantages are introduced by using the application and if the application were capable of being integrated into production services were answered.

The results show that WebTaint helps to combat SQL Injection and Cross-Site Scripting attacks. However, there are drawbacks in the form of additional time and memory overhead. The implemented solution is therefore not suitable for time or memory sensitive domains. A use case for WebTaint is in test environments where security experts utilize the taint tracker to find TaintExceptions through manual or automatic attacks.

Sammanfattning

Internet är en informationskälla och förbinder världen genom en enda plattform. Många företag har utnyttjat detta för att dela information, kommunicera med kunder och skapa nya affärsmöjligheter. Detta kommer dock inte utan nackdelar. Det finns flera potentiella attacker som kan skada webapplikationer.

I avhandlingen implementerades en dynamic taint tracker, kallad WebTaint, för att förhindra sekretess och integritetsproblem i Java-baserade webbapplikationer. Vi utvärderade i vilken utsträckning WebTaint kan bekämpa integritets sårbarheter. Frågorna om vilka fördelar och nackdelar som infördes genom att använda applikationen samt om applikationen kunde integreras i produktionstjänster besvarades.

Resultaten visar att WebTaint hjälper till att bekämpa SQL Injection och Cross-Site Scripting-attacker. Det finns dock nackdelar i form av extra åtgång av tid och minne. Den implementerade lösningen är därför inte lämplig för tids- eller minneskänsliga domäner. Ett användningsfall för WebTaint är i testmiljöer där säkerhetsexperter använder taint trackern för att hitta TaintExceptions genom manuella eller automatiska attacker.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The creation of the World Wide Web has caused a significant impact on today's society [54]. The internet has become an important source of information and it connects the world through a single platform. Many businesses have taken advantage of this to share information, communicate with customers, and create new business opportunities. However, this advancement in technology does not come without drawbacks. The web applications are not only accessible to targeted user groups but also to anyone with access to the web. This enables malicious users to abuse and cause harm to other users of web applications.

There have been a plethora of incidents causing vulnerabilities resulting in everything from money loss, disclosure or destruction of information. One of these incidents is the infamous Heartbleed Bug affecting all users of the OpenSSL cryptographic library. The cryptographic library contained a bug causing protected information to be readable by anybody on the web [46]. Another vulnerability is Stagefright which affected Android users. The vulnerability made it possible through a malicious MMS to gain full control over Smartphones [1].

All applications with accessibility from the web share the problem of managing both trusted and untrusted data. The trusted data comes from the application itself and is, for example, the database. Untrusted data is data modifiable by users through for example an input form. Untrusted data needs to be sanitized before use. The consequences of mistakenly trusting untrusted data can be catastrophic. A variety of tools have been created to minimize the risk of accidentally introduc-

ing security flaws into applications. One of these tools is taint tracking, which attempts to secure an application through information flow control. Taint tracking works by tracking untrusted data through the application into sensitive code areas which for example is database queries. Cases where untrusted data can enter sensitive code areas will be flagged to let the developers know where further development in the form of sanitation is needed to secure the application [32, 49]. This opens for the questions how useful taint tracker is for web applications and if a dynamic taint tracker can be used as a security solution for production services.

1.1 Problem

How can taint tracking secure Java-based web applications? What kind of advantages and disadvantages will this entail?

Code injection is a growing problem and work toward protecting user data is needed. Two of the most common vulnerabilities in the area, according to the Open Web Application Security Project, are Injection attacks and Cross-Site Scripting caused by unsanitized user input [29]. The way to fight the problem with unsanitized user input has many solutions where taint tracking is one. Is it possible to protect web applications by implementing a dynamic taint tracker to run and analyze the code at runtime? To what extent will the taint tracker protect the application? What advantages and disadvantages will it entail? Is the solution capable of being integrated into production services?

1.2 Aim

This thesis aims to implement and evaluate a dynamic taint tracker, called WebTaint, to combat integrity and confidentiality vulnerabilities in Java-based web applications.

WebTaint aims to allow tracking of taint for Strings including data types used for String operations. These data types are String, String-

Builder, StringBuffer, CharArray and ByteArray. WebTaint will be evaluated through case studies and micro-benchmarks to measure the detection rate of vulnerabilities and introduced performance overhead. Concretely, we will implement and benchmark a dynamic taint tracker against SQL Injection and Cross-Site Scripting vulnerabilities.

1.3 Contribution

The contribution of the thesis is continued research in the code injection and information flow field. This is done through:

- Implementing WebTaint. A dynamic taint tracker for Java-based web applications.
- Evaluating WebTaint for vulnerability detection rate and performance overhead.
- Discussing and drawing conclusions regarding usage of dynamic taint tracking for Java-based web applications.

1.4 Limitations

The focus of the thesis lies in security vulnerabilities of web applications. However, other application areas might be subjects to the same kind of vulnerabilities. This thesis will not discuss or present information regarding those areas.

Injection attacks, which is the number one vulnerability on OWASP's top ten vulnerabilities for the year 2017 [29], is a broad vulnerability. This thesis focuses on Injection attacks towards SQL and Cross-Site Scripting. However, detection and prevention of other types of Injection attacks are similar to SQL Injection.

We developed WebTaint for Java-based web applications with the help of the bytecode library Javassist. WebTaint is a dynamic taint tracker constructed to combat integrity and confidentiality vulnerabilities. The evaluation of WebTaint is only conducted on performance overhead and detection of integrity vulnerabilities. This is due to time limitations.

1.5 Methodology

To answer the problem statement in section 1.1, we use a combination of qualitative and quantitative methods. The literature study represents the qualitative research where information about web application security, SQL Injection, Cross-Site Scripting, dynamic taint tracking, and related work is gathered, presented and discussed. This information is needed to comprehend how the taint tracker needs to operate to detect possible malicious user data successfully. The information is gathered from reports and books found through the search portal KTH Primo [51].

The quantitative research consists of the implementation and evaluation of WebTaint where benchmarks for performance overhead and security assurance will be performed. Performance overhead is conducted with with DaCapo [45] benchmark suite consisting of fourteen applications representing common application executions. Security assurance is evaluated through use cases where four applications are tested.

1.6 Ethics & Sustainability

The ethical and sustainability aspects of the thesis have for the majority positive impact. All users of the tool will achieve a gain in some form, except for users with malicious intent. The goal is to combat existing security vulnerabilities in already existing web applications. This increase in security will help companies to provide more secure services. The end users gain is secured information and reduce the risk of becoming victims of code injection attacks.

Usage of WebTaint could result in more robust and secure applications thanks to the ability to find possible security vulnerabilities faster. The taint tracker would as well help the developer in the daily work by validating the soundness of the implemented code. This would ease some of the work for the developer.

The only unethical aspect of WebTaint is the fact that the taint tracker gains access to all data processed by the application. Proper imple-

mentation of the taint tracker is therefore needed to ensure that it is not possible to abuse.

1.7 Outline

The thesis outline starts with the *Background* presenting information regarding the subject. This follows by *Related Work* containing previous research and developed taint trackers. This follows by *Implementation* and *Evaluation* of WebTaint. Next comes *Result* presenting the case study and performance overhead results. Lastly comes the *Discussion*, *Future Work* and *Conclusion* chapters.

Chapter 2

Background

The chapter starts with a general description about Web Application structure. It is followed by a presentation of the CIA Triad commonly used when discussing information security. It is followed by a section about web applications Security Vulnerabilities. Followed by two sections describing Taint Tracking and the programming language Java.

2.1 Web Application

To make applications available and accessible from now days almost everywhere do companies deploy their applications on the web. The deployment of an application can vary a lot, but the most common structure for a web application is based on a three-tier architecture as illustrated in Figure 2.1. The first tier is the presentation tier which contains the visual components rendered by the browser. Logic tier is the second and contains the applications business logic. The third tier is the storage tier, where the business logic stores data as needed [8].

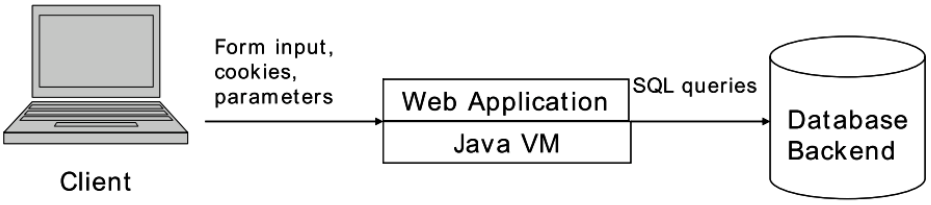

Figure 2.1: An illustration of the three-tier architecture commonly used by web applications [16].

From Figure 2.1 it can be seen that tiers only communicate with the tiers closest to themselves. This causes the logic tier to become a safeguard for the storage tier where valuable and possibly sensitive information is stored. The sensitive information might, for example, be users name, email, personal security numbers and credit card information [8].

The scope of the thesis lies in the logic tier where both trusted and untrusted data is processed, and validations are needed to ensure secure applications. The programming language for the logic tier can vary a lot, but one common and the chosen language for this thesis is Java.

2.1.1 Structured Query Language

Communication between the logic and storage tier is done through a standardized language called Structured Query Language, mostly known as SQL. SQL is created to manipulate and access databases programmatically. The majority of today's database uses SQL. The language works by building queries specifying the required information or task. The query is then evaluated and handled up upon by the SQL engine [10].

2.2 CIA Triad

Discussions regarding information security often rely on the CIA Triad. CIA stands for confidentiality, integrity, and availability as displayed in Figure 2.2. Confidentiality ensures data usage by only authorized individuals. Integrity specifies that application data should be accurate

and unaltered. Availability is the ability to access the application and application data [7].

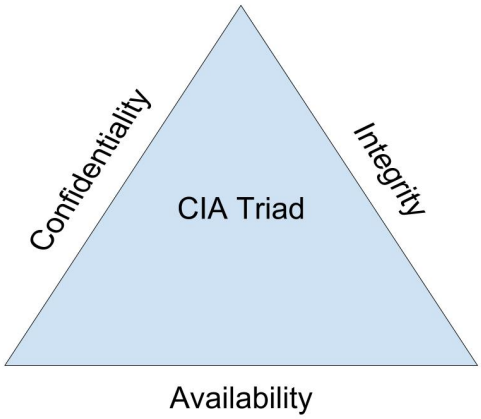

Figure 2.2: An illustration of the CIA Triad, model used when discussing information security.

2.3 Security Vulnerabilities

The organization Open Web Applications Security Project, known as OWASP, is an online community which aims to provide knowledge on how to secure web applications [27]. OWASP has produced reports about the top ten security risks for web applications, and the latest was published in 2017. The report contains information about the ten most common security risks for the given year. Information such as how the security risk is exploited and possible prevention methods are also presented. This thesis will look at security risk number one and seven from the latest report. These two security risks are vulnerabilities of information disclosure and code injection. The two vulnerabilities are Injection attack and Cross-Site Scripting [29].

2.3.1 SQL Injection Attacks

The most common security risk is Injection Attacks [29]. Injection Attack is an attack where the attacker's input changes the intent of the

execution. The typical results of Injection Attacks are file destruction, lack of accountability, denial of access and data loss [42].

Injection Attacks exists towards a broad set of different areas but the area discussed and analyzed in this thesis is SQL Injections. SQL Injections can be divided into two different subgroups. These two subgroups are SQL Injection and Blind SQL Injection [42].

SQL Injection

SQL Injection occurs when an SQL query is tampered with which results in gaining content or executing a command on the database which was not intended. Listing 2.1 displays an SQL query which is open to SQL Injections. This is because the variable UserId is never validated before it is propagated into the query [8, 42].

Listing 2.1: Pseudo code acceptable to SQL Injection through malicious usage of *userInput*.

```
userId = userInput
"SELECT * FROM Users WHERE userId = " + userId
```

The query work as intended if the user input, labeled as *userInput*, is a valid Integer (since Integer is what we have decided that user id is in the application). An example of malicious usage of user input is *10 or 1 = 1*. This input would result in the query seen in Listing 2.2.

Listing 2.2: An example of SQL Injection where the whole Users table is returned

```
SELECT * FROM Users WHERE userId = 10 or 1 = 1
```

This query results in an execution that always evaluates to true. As a result, the query returns the whole table of users. This problem can be prevented in different ways. The first is through validation of input.

By verifying user input as in Listing 2.3 can we protect the query from being vulnerable to the SQL Injection.

Listing 2.3: An example of SQL Injection prevention through variable sanitiazion.

```
userId = userInput
isInteger(userId)
"SELECT * FROM Users WHERE userId = " + userId
```

A second common alternative is to use SQL Parameters which handle the verification for the user. This leaves the verification and validation of input up to the SQL engine. An example written with SQL Parameters can be seen in Listing 2.4.

Listing 2.4: An example of SQL Injection prevention through SQL Parameters.

```
userId = userInput
sqlQuery = "SELECT * FROM Users WHERE userId = @0"
db.Execute(sqlQuery, userId)
```

Blind SQL Injection

Blind SQL Injection is very similar to SQL Injection. The only difference is that the attacker does not receive the requested information in clear text from the database. The information is instead received by monitoring variables such as how long time the response takes or what kind of error messages it returns. An example of the first is an SQL query that tells the SQL engine to sleep depending on a condition. An example of this can be seen in Listing 2.5 [8, 42].

Listing 2.5: An example of Blind SQL Injection where query response is delayed five seconds if a user with id one is in the Users table.

```
SELECT * FROM Users WHERE userId = 1 WAITFOR DELAY
    '0:0:5'
```

The second variant of a Blind SQL Injection is through analyzing error messages and, depending on what they return, build an image of the targeted data. This is mostly done through testing different combinations of true and false queries [8, 42].

2.3.2 Cross-Site Scripting

Cross-Site Scripting has been a vulnerability since the introduction of JavaScript in websites. One of the first Cross-Site Scripting attacks was carried out just after the release. The attack was conducted through loading a malicious web application into a frame on the site that the attacker wants to gain access to. The attacker could then through JavaScript access any content visible or typed into the web application. Same-Origin Policy was introduced to prevent this form of attacks. The policy restricts JavaScript to only access content from its origin [13, 37].

The introduction of the Same-Origin Policy, however, did not stop the attackers. The next wave of attacks was mostly towards chat rooms where it was possible to inject malicious Cross-Site Scripts into the message input form. Which would then be reflected by the server itself, when displaying the message for other users, and thereby bypassing the Same-Origin Policy [13].

There are three different types of Cross-Site Scripting. These three are reflected, stored, and DOM-based Cross-Site Scripting.

Reflected Cross-Site Scripting

Reflected Cross-Site Scripting, mostly conducted through a malicious link that a user accesses. The malicious link will exploit a vulnerable input on the targeted web application and through the input reflect malicious content to the user [42].

Stored Cross-Site Scripting

Stored Cross-Site Scripting is when malicious scripts get stored in the targeted web applications database. This malicious script is then

loaded and presented to each user who is trying to access the application [42].

DOM-based Cross-Site Scripting

DOM-based Cross-Site Scripting is very similar to Reflected Cross-Site Scripting, but it does not necessarily have to be reflected from the application server. DOM-based Cross-Site Scripting modifies the DOM tree, and through that, it exploits the user [42].

2.4 Taint Tracking

Taint tracking, also known as taint analysis, is a tool to analyze the flow of information in an application [32]. The goal of taint tracking is to prevent possible attacks such as Injection and Cross-Site Scripting by enforcing the usage of sanitizers on input data. Taint tracking can be implemented in two different forms: static and dynamic. Static taint tracking is an evaluation tool possible to be included in the integrated development environment where it notifies the developer of possible security vulnerabilities. Dynamic taint tracking is a tool used simultaneously as the application execution. Dynamic tracking analyses for vulnerabilities at runtime and achieve higher accuracy compared to static tracking. The advantage for static is the ability to execute before runtime, but its disadvantage is the lower accuracy in tracking of taint.

Taint trackers operate by tracking untrusted data and acting upon any that are trying to enter sinks without first being sanitized. Perl and Ruby are two programming languages which have adapted to use taint checking [33, 24]. There are some tools which enable taint checking for other platforms. TaintDroid [25] for the Android platform is one of them. This thesis will discuss dynamic taint tracking and how it can improve the security of Java-based web applications.

Taint tracking contains four main steps which are described in Table 2.1. The first step is marking all data from untrusted sources as tainted. This is done through a taint flag attached to the input variables. Step two is tracking taint where tainted data propagates its tainted flag onto

all data it comes in contact with. The third step is the possibility of detainting data, but this is only done after the data have been sanitized through predefined sanitizers. The fourth and last step is checking the taint flags in areas called sinks which are entry points to sensitive code [32, 49]. The decision of what to do if a tainted variable tries to pass through a sink varies depending on the application. However, remedial actions should be conducted. These actions should be, depending on the application owners choice, logging the events, throwing an error, or modifying the tainted values into safe predefined values.

Table 2.1: The four core tasks behind taint tracking.

Tainting	Marking all data from sources as tainted.
Propagat Taint	Propagating taint to all data coming in contact with tainted data.
Detainting	Marking all data from sanitizers as non-tainted.
Assert Non-tainted	Assert that data passing through sinks are non-tainted.

An example of taint tracking can be seen in Listing 2.6. In this example *getAttribute* is a source, *executeQuery* is a sink and *validate* is a sanitizer. On line 1, the input from the source is flagged as tainted, and the taint propagates onto *userId*. The sanitizer on line 2 validates *userId* and removes the tainted flag. Lastly, the sink on line 3 executes the query since the argument is not tainted. If a user sends in a malicious userId containing ”101 OR 1 = 1” the validator would sanitize the String and safely execute the sink command. However, removing line two would result in tainted data entering the sink. Without a dynamic taint tracker this would result in giving the malicious user the entire list of Users. With a dynamic taint tracker, however, the result is the sink halting the execution, therefore, preventing unwanted information disclosure.

Listing 2.6: A code example of accurately handling user input before accessing sensitive code area.

```
1 userId = getAttribute("userId");
2 validate(userId)
3 executeQuery("SELECT␣*␣FROM␣Users␣WHERE␣userId␣=␣"
    + userId);
```

2.5 Java

Java has been around since the early 90's. The founder's objective was to develop a new improved programming language that simplified the task for the developers but still had a familiar C/C++ syntax. [28]. Today is Java one of the most common programming languages [14].

Java is a statically typed language which means that no variable can be used before being declared. The variables can be of two different types: primitives and references to objects. Among the primitive types does Java have support for eight. These are byte, short, int, long, float, double, boolean and char [35].

2.5.1 Java Virtual Machine

There exist a plethora of implementations of the Java Virtual Machine, but the official developed by Oracle is HotSpot [47]. One of the core ideas with Java during its development was "Write once, run anywhere." The slogan was created by Sun Microsystems which at the time were the company developing Java and the Java Virtual Machine. [9]. The idea behind the Java Virtual Machine was to enable one language to be platform independent and then modify the Java Virtual Machine to run on as many platforms as possible. The Java Virtual Machine is a virtual machine with its own components of heap storage, stack, program counter, method area, and runtime constant pool.

Figure 2.3 illustrates the architecture of the Java Virtual Machine. The Class Loader loads the compiled Java code and adds it into the Java Virtual Machine Memory. The Execution Engine reads the loaded byte-

code from the Java Virtual Machine Memory and executes the application instructions. The Java Virtual Machine has built-in support for Java Agents which is a tool running between the Java Virtual Machine and the executed Java application. An Agent is loaded and given access to the application by the Class Loader. The Class Loader will trigger the implemented Java Agent and allow for instrumentation of each class file loaded by the Class Loader before being loaded into the Java Virtual Machine [50, 22].

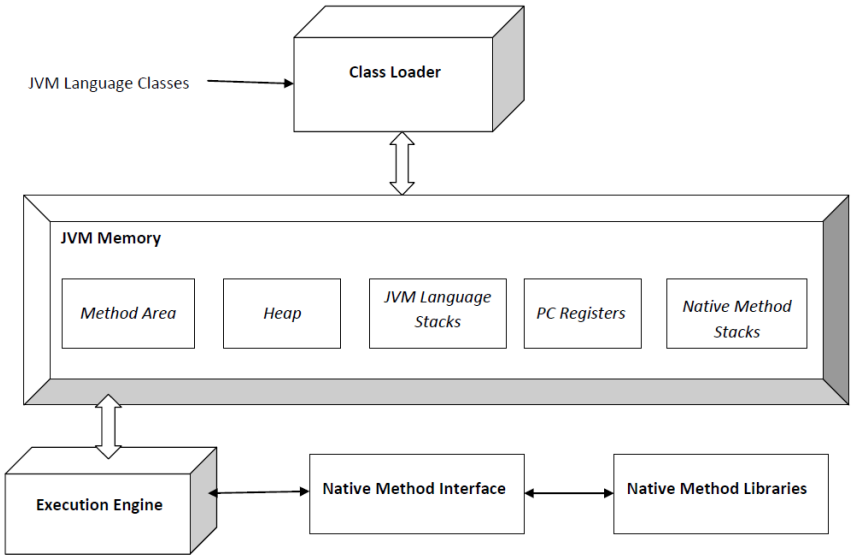

Figure 2.3: An illustration of the Java Virtual Machine Architecture [21].

2.5.2 Instrumentation

Java instrumentation is a way to modify the execution of an application without knowing or modifying the application code itself. Good use cases for Java instrumentation are, for example, monitoring agents, event loggers, and taint trackers. Instrumentation is an official Java package that provides services needed to modify the bytecode of program instructions. It is conducted through implementing an Agent that makes it possible to transform every class loaded by the Class Loader before being used for the first time. However, there is a library of classes which cannot be instrumented by an Agent. This library is the

rt.jar containing the Base Java Runtime Environment which is needed to start up the Java Virtual Machine including the Class Loader. Instrumentation of the Base Java Runtime Environment needs to be done before running the Java application.

The Java Agent operates on bytecode which is time-consuming work for the developer. To ease the task of instrumentation is the bytecode instrumentation library Javassist used [20, 23].

2.5.3 Javassist

There exist several libraries that can help the developer in the task of creating a Java Agent. The help comes in libraries of methods to manipulate Java bytecode. The library used in this thesis is Javassist. Javassist stands for Java programming Assistant and provides two levels of API. The two are on source respectively bytecode level. We used the source level API which is providing the functionality of manipulating Java bytecode with little bytecode knowledge [23].

The Javassist source level API provides classes representing instances of classes, methods, and fields. These API classes contain methods to use when computing if the given class, method or field should be instrumented. The classes representing methods do also contain the methods *insertBefore*, *insertAfter* or *insertAt*. The three methods allow inserting Java code to the beginning, the end or at a specific position of the method.

Chapter 3

Related Work

This chapter presents the related work in the field. It aims to handle the areas of dynamic and static taint tracking for Java as well as other related platforms.

Haldar, Chandra, and Franz [16] has written a report about dynamic taint tracking for Java applications where they tried to solve the problem of not correctly validating user input. They managed to construct a tool that is independent of the web applications source code and the results from using the tool is gain in security. Haldar, Chandra, and Franz [16] ran their benchmarks on OWASP's project WebGoat [6] but acknowledged in their report that benchmarks of real-world web applications need conducting. Their application implemented taint tracking for Strings by adding a taint flag and altered the methods to propagate the taint into the String class file. Haldar, Chandra, and Franz [16] implementation cant be found.

Another implementation of a dynamic taint tracker for Java applications, of a tool believed to be the current state of the art, is Phosphor [34] created by Bell and Kaiser [5]. Phosphor developed with the help of the Java bytecode manipulation library ASM [3]. The application solved tracking of taint for primitives and arrays by introducing shadow variables. A shadow variable is a variable holding the taint value for a un-instrumentable object. The shadow variable is instrumented into the application and placed next to each primitive and array. Each method in the application is also instrumented to pass shadow variables together with the un-instrumentable object. Phosphor does however not

support detainting of sanitized variables which makes it mostly used for specifying specific data flows to analyze [5].

A third implementation of a dynamic taint tracker for Java applications is Dynamic Security Taint Propagation [11] constructed with the help of the Java library AspectJ which enables aspect-oriented programming in Java [43]. Dynamic Security Taint Propagation only propagates taint for the String, StringBuffer, and StringBuilder classes. The tracking works by creating aspect-oriented events that trigger the tracking of taint, tainting sources, and assertions that ensure that tainted values do not pass through sinks.

Except for dynamic taint trackers for Java applications does it as well exist dynamic taint trackers for other platforms. One of these is TaintDroid which is constructed for Android smartphones and aims to prevent information theft [12]. TaintDroid uses what they call shadow memory for their implementation approach which they claim to reduce the memory overhead by the application. Hsiao et al. [17] has as well conducted further work on top of TaintDroid where they created a security scheme called PasDroid. PasDroid enabled the users to gain full control over the management of possible information leaks.

It does as well exist static taint trackers where FlowDroid is one of these. Arzt et al. [2] created FlowDroid which computes data flows for Java and Android applications. Another application for Java and HTML applications is HybridFlow [18] and an application statically tracking taint for Python-based web applications is Python Taint [36].

Chapter 4

Implementation

This chapter presents the fundamental parts of the implementation process of WebTaint. The chapter starts with a section describing Policies enforced by the application. This section is then followed by how Sources, Sinks & Sanitizers was specified and lastly comes the WebTaint architecture.

4.1 Policies

The development of WebTaint relies on the tasks described in Table 2.1 in chapter 2. These are tainting, detainting, propagating taint, and assertion of non-tainted data. However, to implement the functionality behind the taint tracker needs security policies first be defined. Security policies are principles or actions that the application strives to fulfill [4]. The taint tracker developed in this thesis will base on two different aspects. These are *integrity* and *confidentiality*.

4.1.1 Integrity

The integrity policy defines that users may not modify data which they do not have permission to alter. The goal is to ensure prevention of malicious usage of the application where attackers aim to inject malicious executions that can, for example, lead to information disclosure or destruction of application data. To ensure this is the following policy defined:

- No information or execution of code shall be altered without the users having permission to do so for the given information or execution.

This entails that no information from sources shall pass through a sink without first being sanitized.

4.1.2 Confidentiality

The confidentiality policy defines that data given to the user should only be data that the user have the right to access. The goal is to ensure prevention of malicious usage of applications where attackers aim to steal application data. This gives us the policy:

- No information shall be released to users without the user having the correct permission for the given information.

This entails that no information from sources shall pass through a sink unless it has the permission to do so.

While monitoring confidentiality vulnerabilities are the sources and sinks roles reversed. Meaning that the information to protect is marked as sources and sinks are the systems exit points. The sanitizers are used to specify non-malicious sinks.

4.1.3 Taint Checking

The integrity policy is enforced by making sure validations, of user input and any data it has come into contact with, are conducted. By enforcing these should preventions of integrity volubilities be reduced severely. The confidentiality policy above is enforced by validating that data from specified sources do not pass through any sink except for those specified as sanitizer as well.

The policies above can be translated into taint policies. These are:

- Data passing through sources, going into the domain, shall always be marked tainted.
- Tainted data is never allowed to pass through sinks.
- Predefined sanitizers are the only method calls allowed to detaint data.

4.1.4 Taint Tracking

Taint checking policies, however, are not the only policies needed to implement WebTaint. WebTaint need policies specifying how the taint tracking should operate. Below are rules defining when taint should propagate:

- Data resulting in a copy, subset or combination of tainted data shall flag as tainted.
- Data disclosing information about tainted data shall flag as tainted.

4.2 Sources, Sinks & Sanitizers

Defining the source, sinks, and sanitizers is a large task in itself. There is no official documentation in Java specifying these and depending on the application, framework and library used might this vary a lot. The sources, sinks and sanitizers used in this thesis is an aggregation from *Which methods should be considered "Sources", "Sinks" or "Sanitization" ?* [52] and *Searching for Code in J2EE/Java* [38]. These web pages present sources, sinks, and sanitizers from their experience with developing web applications.

4.3 WebTaint

The implementation of WebTaint is divided into three subprojects. These three are Agent, Xboot, and Utils.

Agent Running at runtime and triggers the transformation of sources, sinks, and sanitizers.

Xboot Running prior runtime and transforms sources, sinks, sanitizers, String, StringBuilder, and StringBuffer in the base Runtime Environment.

Utils Utilities to transform classes into sources, sinks, and sanitizers.

The reasoning behind the division is because of the need of transforming classes both before runtime and during runtime. The Agent is handling the transformation in runtime and Xboot transforms classes on command before runtime. The logic of transforming the classes is, however, the same in both projects and centralized in the Utils project.

Transformation of the String, StringBuilder, and StringBuffer classes, to support tracking of taint, is done in the Xboot project. It is done by iterating through each class and transforming them to: contain a variable to hold the taint flag, and propagate the taint in each method and constructor. The propagation of taint happened when either the taint flag of the class or any of the arguments marked tainted.

The architecture of WebTaint is seen in Figure 4.1. The Java Virtual Machine is un-instrumented and loads WebTaint through the Class Loader at startup. WebTaint gets access to every class file loaded and instruments the web applications and all needed libraries.

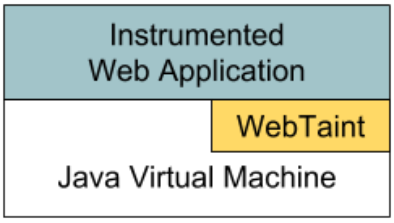

Figure 4.1: Architecture of WebTaint

4.3.1 The Utils Project

The Utils project includes the core logic of marking methods as sources, sinks, and sanitizers. The major part of the implementation is the same for sources, sinks, and sanitizers and the only difference is how they get instrumented if a source, sink or sanitizer is found. The logic behind finding the classes to instrument works by taking all class and matching them with the three criteria below.

- Is same class as a defined source, sink or sanitizer.
- Implements interface of a defined source, sink or sanitizer.
- Extends a defined source, sink or sanitizer class (recursive).

The last item is a recursive call where each extends class is analyzed if it qualifies for any of the three criteria. If a class fulfills, any of the three criteria, are the class instrumented.

The instrumentation of the class works by iterating through each class method and instrumenting them depending on the matching type. Instrumentation of sources will set the return parameter of the method as tainted. Instrumentation of sanitizers works by detainting the return value of the method. For sinks will an assertion call check that none of the method arguments are tainted. If anyone of them is, then a taint exception occurs, and remedial actions conducted. This action should be, depending on options, a logging event, throwing an error or modifying the tainted value into a safe predefined value. During the

conducted benchmarking is the option of a predefined value, where the value is an empty string, and logging the event used.

4.3.2 Notable Problems

One of the first problems that were introduced during the development of the application was that some classes could not be instrumented during runtime. More precisely, the classes that the Java Virtual Machine relies on can't be instrumented at runtime. However, there is a solution to this. The solution is to pre-instrument the base Java Runtime Environment and create a new instrumented rt.jar file with statically modified versions of the classes. The created jar file loads through the option *Xbootclasspath/p* that appends the classes to the front of the bootstrap classpath. Making the Java Virtual Machine use our modified versions of the base Java Runtime Environment [19] before the original version.

Another problem is that instrumentation of primitives and arrays not possible. It causes a problem since it opens the ability to miss tracking of tainted data if String operations are done with byte- or char arrays. The solution that can solve this is to create shadow as Bell and Kaiser [5] did while creating Phosphor [34]. However, other possible solutions such as creating a centralized memory bank just as Enck et al. [12] did when implementing TaintDroid. They, however, called this shadow memory.

Another problem that emerged was that operations with primitives are direct bytecode translations. Two examples of these are the usage of + (addition) and - (subtraction). Adding operations to these through Javassist's source level API is therefore not possible. To solve this are operations on bytecode level needed. [23].

Chapter 5

Evaluation

This chapter describes the evaluation of WebTaint. The chapter starts with a description of the Test Environment followed by a detailed description of the Benchmarking.

5.1 Test Environment

The benchmarks are conducted on an Asus Zenbook UZ32LN, bought in spring 2015. No other programs were running while benchmarks were in process. The specifications of the computer and other important metrics are the following:

Processor: 2 GHz i7-4510U

Memory: 8 GB 1600 MHz DDR3

Operating system: Ubuntu 17.10

Java: OpenJDK 1.8.0_162

Java Virtual Machine: OpenJDK 25.162-b12, 64-Bit, mixed mode

5.2 Benchmarking

The benchmarks conducted are on applications, where we will look at security gain, and on a performance benchmarking suit where time

and memory overhead was examined. To evaluate the security gain and performance overhead is every benchmark executed twice. The first time is without WebTaint activated and the second with WebTaint activated. The reason behind this is first to acquire the baseline values which then can be used to compare with the WebTaint values. The difference between these is the security gain and performance overhead.

5.2.1 Applications

To detect security vulnerabilities in the applications has OWASP Zed Attack Proxy [31] known as ZAP ben used. ZAP is an open-source security scanner for web applications which is widely used in the penetration testing industry. Usage of ZAP is seen in Figure 5.1. ZAP accesses the web application over HTTP(S) communication and tries to find vulnerabilities in the applications wherever user input is possible.

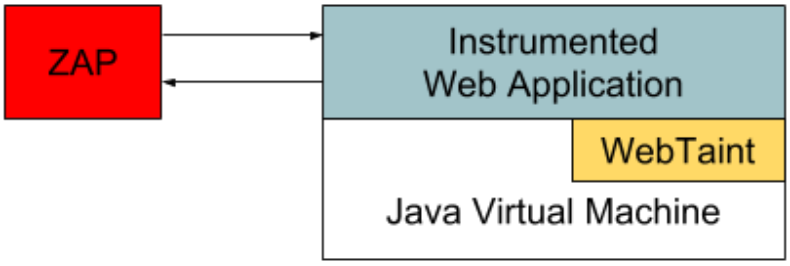

Figure 5.1: Evaluation architecture of ZAP and WebTaint

Extensive ZAP scans are very time-consuming, and to the only scan, the applications for vulnerabilities of interest is a new policy specified in ZAP. The policy is modified only to contain the Injection category where the tests in Table 5.1 are used.

Table 5.1: Security Vulnerabilities Detected by WebTaint in Ticketbook

- Buffer Overflow
- CRLF Injection
- Cross-Site Scripting (Persistent)
- Cross-Site Scripting (Persistent) - Prime
- Cross-Site Scripting (Persistent) - Spider
- Cross-Site Scripting (Reflected)
- Format String Error
- Parameter Tampering
- Remote OS Command Injection
- SQL Injection

Every scan of an application starts with spidering to detects all possible entries to the system. If the application requires authentication to access parts of the application is the needed credentials added to the ZAP context, and then execute the spider the application once again. After these steps are the scanning of the application started and possible vulnerabilities found are stored in log files.

The benchmarking of applications are a very time-consuming task and therefore only conducted on four applications. Each application is Java-based, and all of them are containing security vulnerabilities including one or more of SQL Injection attacks or Cross-Site Scripting. These four Java web applications are presented in the sections below.

Stanford SecuriBench Micro

Stanford SecuriBench Micro is a set of small test cases designed to evaluate security analyzers. The test suit deliberately insecure and was created as part of the Griffin Security Project [15] at Stanford University and contains 96 test cases and 46407 lines of code. This thesis uses version 1.08 of the application and contains the vulnerabilities SQL Injection, Cross-Site Scripting, HTTP Splitting, Path Traversal and more [40, 41].

InsecureWebApp

InsecureWebApp is a deliberately insecure web application developed by OWASP to show possible security vulnerabilities and what harm they can cause to a web application. The project consists of 2913 lines of code and version 1.0 is used and contains the vulnerabilities Parameter Tampering, Broken Authentication, SQL Injection, HTML Injection and more [30].

Ticketbook

Ticketbook is deliberately insecure web application developed by Contrast Security to show the power of one of their security tools. The application consist of 13849 lines of code and version 0.9.1-SNAPSHOT is used and contains the vulnerabilities Cross-Site Scripting, Command Injection, Parameter Tampering, XML External Entity and more [48, 26]

SnipSnap

SnipSnap is a Java-based web application developed to provide the necessary infrastructure to create a collaborative encyclopedia. The web page functionality is similar to Wikipedia [53] where users can sign up and contribute by writing posts. The application consists of 566173 lines of code and version 1.0-BETA-1 is used in this thesis [39]. The application is not deliberately insecure, and the policies which it aims to fulfill is the same as specified in Subsection 4.1.1 and 4.1.2. Which is to prevent unauthorized information access, and execution of code as well as preventing information disclosure.

5.2.2 Performance Overhead

To evaluate the time and memory overhead is The DaCapo Benchmark Suit [45] used. DaCapo is a set of applications constructed specifically for Java benchmarking. This thesis uses version DaCapo-9.12-bach which consists of fourteen real-world applications. Table 5.2 contains a description for each application. Summary is taken from DaCapos website [44].

Table 5.2: Descriptions for each application in The DaCapo Benchmark Suit taken from *The benchmarks* [44]

Avrora Simulates a number of programs run on a grid of AVR microcontrollers.

Batik Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.

Eclipse Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.

Fop Takes an XSL-FO file, parses it and formats it, generating a PDF file.

H2 Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark.

Jython Interprets a the pybench Python benchmark.

Luindex Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.

Lusearch Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.

Pmd Analyzes a set of Java classes for a range of source code problems.

Sunflow Renders a set of images using ray tracing.

Tomcat Runs a set of queries against a Tomcat server retrieving and verifying the resulting web pages.

Tradebeans Runs the daytrader benchmark via a Jave Beans to a GERONIMO backend with an in-memory h2 as the underlying database.

Tradesoap Runs the daytrader benchmark via a SOAP to a GERONIMO backend with in-memory h2 as the underlying database.

Xalan Transforms XML documents into HTML.

The measurement of time and memory is conducted through a C script constructed to execute each application DaCapo application ten times, both with and without WebTaint activated. To isolate each iteration is a unique process spawned per test execution. This process runs the benchmarked application in a child process and monitors the time and memory of the execution. The information is then sent back to the main thread where all data is processed and aggregated.

Chapter 6

Result

This chapter presents the results of the conducted evaluation. Appendix A contains raw data and metrics of data that may not be shown in this chapter. The chapter starts with presenting the results from the Applications evaluation where Java-based web applications have been evaluated measuring security vulnerabilities with and without WebTaint. The results from the Performance Overhead evaluations then follow it where the parameters time and memory is measured.

6.1 Applications

The presented results in this section are from evaluating Java applications for security vulnerabilities with and without dynamic taint tracking. The results from each application are listed in its table where vulnerability type and the number of vulnerabilities are listed. In the presentation of the result in the text are vulnerabilities of the same type aggregated. By aggregating all four average prevention rates do we get that the overall prevention rate for the applications is 81%.

Table 6.1 shows the vulnerabilities from evaluating Stanford SecuriBench Micro [40]. In the table can we see that the most common vulnerability is reflected Cross-Site Scripting where 71 vulnerabilities are present. Second most common is SQL Injection with 20 and the least common with one vulnerability is Buffer Overflow. By enabling

dynamic taint tracking on the Stanford SecuriBench Micro [40] application results in a 100% prevention rate.

Table 6.1: Security Vulnerabilities Detected by WebTaint in Stanford SecuriBench Micro

	Vulnerabilities	Found by WebTaint
Cross-Site Scripting (Reflected)	71	71
SQL Injection	20	20
Buffer Overflow	1	1

Table 6.2 shows the vulnerabilities from running InsecureWebApp [30] with and without dynamic taint tracker. Of the two types of vulnerabilities is SQL Injection the first with six vulnerabilities and reflected Cross-Site Scripting with two. Enabling dynamic taint tracking on InsecureWebApp [30] results in 100% prevention rate on SQL Injection attacks and 0% for Cross-Site Scripting. The overall prevention rate is 75%.

Table 6.2: Security Vulnerabilities Detected by WebTaint in InsecureWebApp

	Vulnerabilities	Found by WebTaint
Cross-Site Scripting (Reflected)	2	0
SQL Injection - Authentication Bypass	2	2
SQL Injection - Hypersonic SQL	4	4

Table 6.3 shows the vulnerabilities from evaluating Ticketbook [48]. The most common vulnerability was Cross-Site Scripting with 14 occurrences. SQL Injection was the least with one. The prevention rate of SQL Injection was 100% and for Cross-Site Scripting 71.4%. The overall prevention rate is 73.3%.

Table 6.3: Security Vulnerabilities Detected by WebTaint in Ticketbook

	Vulnerabilities	Found by WebTaint
Cross-Site Scripting (Persistent)	2	2
Cross-Site Scripting (Reflected)	12	8
SQL Injection	1	1

The results from evaluating the application SnipSnap [39] is seen in Table 6.4. In this table can we see that the most common vulnerability is reflected Cross-Site Scripting with 172 occurrences. Second Largest is SQL Injection with 49 occurrences followed by CRLF Injection with two. Enabling dynamic taint tracking yields an overall prevention rate of 77.2%. All CRLF Injection is prevented. Cross-Site Scripting prevented with 77.3% and SQL Injection with 75.5%.

Table 6.4: Security Vulnerabilities Detected by WebTaint in SnipSnap

	Vulnerabilities	**Found by WebTaint**
Cross-Site Scripting (Reflected)	172	133
CRLF Injection	3	3
SQL Injection	47	37
SQL Injection - Authentication Bypass	2	0

6.2 Performance Overhead

The results from benchmarking the application on DaCapo Benchmark Suit [45] is seen in Figure 6.1 and 6.2. Both graphs are constructed to show the added overhead of running the applications with dynamic taint tracking activated. The graphs are constructed based on the data in Table A.1 and A.2.

6.2.1 Time

Figure 6.1 displays the results of the average time overhead per application. The results show that the application with the least average time overhead was Tradesoap where 14.7% was added. The largest application, however, was Batik with an overhead of 432.2%. The average overall is 162.9%.

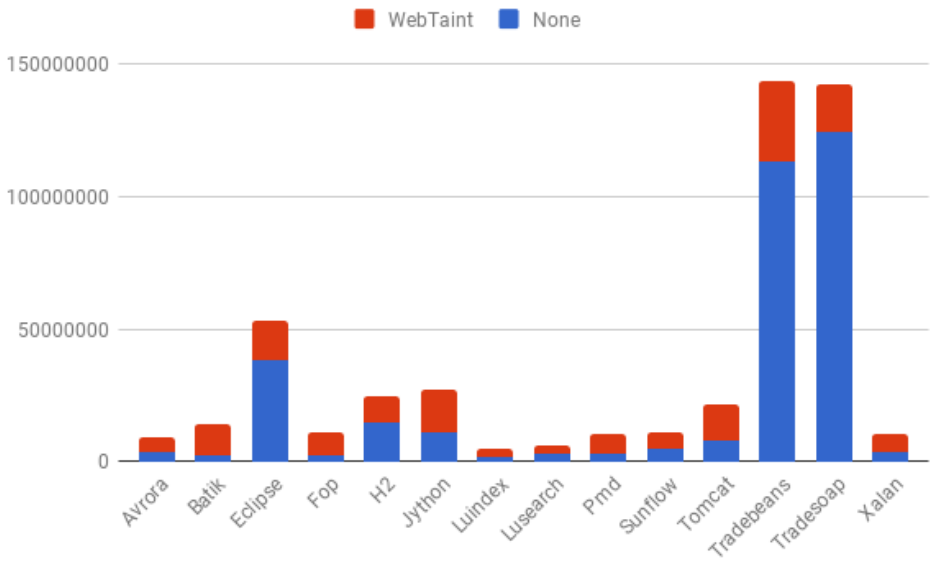

Figure 6.1: Average Added Time in Microseconds

6.2.2 Memory

Figure 6.2 displays the results of the average memory overhead per application. The results show that the application with the least average memory overhead was Eclipse where 5.5% was added. The largest application, however, was Batik with an overhead of 344.6%. The average overall is 142.7%.

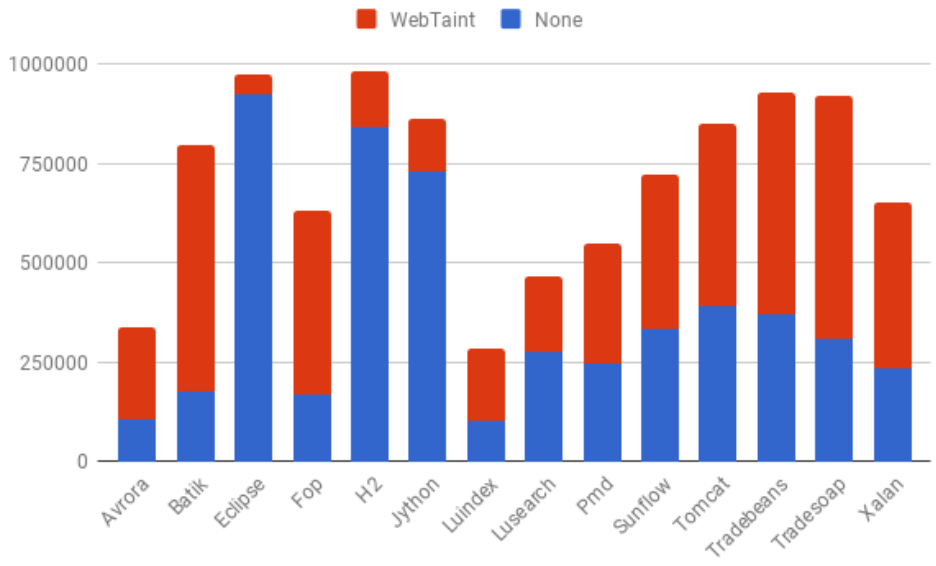

Figure 6.2: Average Added Memory in Kilobytes

Chapter 7

Discussion

This chapter contains the discussions about the implemented dynamic taint tracker, named WebTaint, and how well it performs. The chapter starts with a general discussion about the implementation and the result. This is then followed by Taint Tracking discussions. Lastly is there two sections about Sources, Sinks & Sanitizers and Methodology of Evaluation.

By looking at the results, of evaluating applications with and without WebTaint activated, do we for all four applications see a significant improvement. Stanford SecuriBench Micro have a 100% prevention rate by using WebTaint, and the other three applications have 75.5%, 75%, and 73.3%. Making the average across the four to be 81%. It is a significant impact in combating integrity vulnerabilities. Further development implementing support for taint tracking of more data types could even increase this number.

However, gain in security might not be worth it if it comes with significant drawbacks as well. From the results can we see that using WebTaint adds a performance overhead. This overhead comes from instrumenting classes and the added operations to tracking taint. The most significant impact on time overhead comes from the instrumentation phase which can be seen in both Figure 6.1. The instrumentation happens the first time each class is loaded into the Java Virtual Machine. It means that applications with longer runtimes are less impacted by the added time overhead. It is seen by comparing the execution time of Avrora and Batika, 137.2% respectively 432.2%, with Tradebeans and Tradesoap, 26.3% respectively 14.7%.

Memory overhead tells a different story. The two most extended executions, together with almost every execution have the same memory overhead as the average which is 142.7%. It is only the Eclipse, H2 and Jython DaCapo tests that have significantly less. It is hard to say why these are significantly less. One guess would be that they are significantly smaller in usage if Strings and therefore not affected by the added taint flag and help functions appended to the String, StringBuilder, and StringBuffer classes.

Both time and memory overhead could more than probably be optimized by reworking the implemented code to be more effective. This was however not conducted in this thesis because of time issues.

Acceptance of performance overhead is never a good idea if nothing good comes out of it. Though, an average prevention rate of 81% is something that could make overhead acceptable to the system. By further optimizing the underlying logic and enhancing support for more data types would make the gain compared to the cost make it even more worth it. However, this could even after optimizations not be a possible use case for time and memory sensitive domains. It might even be possible to impair the user experience of a web application if the added time for events to happen takes a long time. An optimal solution for this might instead be not to use WebTaint as a security solution for the production service but instead use it on the staging server where flow tests of the servers are automatically run to test trigger any taint exceptions.

7.1 Taint Tracking

Due to time issue were only the classes String, StringBuilder and StringBuffer implemented to supports taint tracking. These are the most important classes for taint tracking when securing web applications since all inputs are once in the form of Strings. However, there is a risk of losing the tracking of taint since some libraries use char and byte arrays for String operations.

However, the results prove that the implemented classes do cause a significant difference. Nevertheless, the optimal solution would be with

complete integration for all data types in Java. Just like Phosphor, but with the ability to sanitize variables.

7.2 Sources, Sinks & Sanitizers

One part of the thesis, believed in the beginning to be a minor part, was defining sources, sinks, and sanitizers. The implementation of taint trackers is heavily dependent on the defined sources, sinks, and sanitizers. Much work is needed in this area to define these correctly. However, since time was an issue, throughout this thesis, was the solution to aggregate definitions of sources, sinks, and sanitizers from the few works done by others.

The optimal solution, however, would be extensive research where lists for each library, framework, and deployment utility commonly used by Java-based web applications was compiled. These lists could then be subscribed on depending on what type of application is used. Best case would be that every developer of Java libraries, frameworks, and deployment utilities compiled lists for their implementations.

Another thing of interest would be to introduce multiple taint types. It would be used to ensure data from sources to be sanitized with the correct sanitizer depending on the source or sink type. The reason behind this is because data from one source type might not be possible to sanitize with all types of sanitizers. It might also be that some sinks need a particular type of sanitation to work safely.

The question of what to do when a taint exception gets caught is an interesting question. We believe that this depends a lot depending on the ”mode” that the owner wants the application to execute in. For the ”lightest” mode is logging sufficient. Telling what kind of taint exception occurred enables the application to get corrected at a later time. At a higher mode should an exception be thrown or predefined values used. This prevents the possible malicious user input to execute.

The remedial action could even be to sanitize the data with a predefined sanitation function depending on the source or sink type. It could be as good as it is terrible. It could go two different ways. The first is that this would help prevent some vulnerabilities through automatic

sanitation. The developer does than patch the problem by adding correct validation where it is missing. The second possibility is the same example except that the developer does not patch the problem. Also, maybe even lowers the ambition of correctly validating user input in future since the taint tracker can do some sanitization. We believe however that we are far away from implementing a taint tracker that is smart enough to take over the role of sanitizing user variables.

7.3 Methodology of Evaluation

The thesis objective was from the beginning to implement the dynamic taint tracker, WebTaint, and benchmark it in comparison to Dynamic Security Taint Propagation and Phosphor. This was sadly not possible since the prior was not possible to build from the source files and Phosphor do not support sanitation of variables. Making the use case for Phosphor not applicable in comparison to the implemented taint tracker. It is hard to say how well the implemented tool performed when a comparison was not possible. However, the results prove that the implementation could be of use.

Chapter 8

Future Work

There is work needed to be done before WebTaint can take place as a solution to secure web applications. One of these is to conduct comprehensive work regarding sources, sinks, and sanitizers to ensure correct usage. It would also be of interest to implement the use of different types of sources, sinks, and sanitizers. Allowing for an advanced taint tracking where sanitizers only capable of sanitizing a specific type of data do not mark other types as safe.

Optimizations of WebTaint is also in need since it was not prioritized during the thesis. Improvements minimizing the added overhead would make WebTaint usable in time- or memory sensitive domains. WebTaint would also benefit by enhancing the coverage of data types supporting taint tracking. The two most important data types, not supporting taint tracking in WebTaint, are char and byte arrays.

Another possible WebTaint enhancement is implementing support of implicit taint flows. WebTaint does at the moment only support explicit flows where taint propagates if the calculated variables are directly dependent on a tainted variable. For example would x in $x = y + 1$ become tainted when y is tainted. The implicit flow would enable taint to propagate implicitly. A example of this is that x would be tainted in $if(y)x = 1$ when y is tainted.

Chapter 9

Conclusion

We implemented and evaluated a dynamic taint tracker for Java-based web applications called WebTaint. The goal of the tool was to combat integrity and confidentiality vulnerabilities. The results of the conducted evaluations show improved security when using WebTaint. However, there are drawbacks regarding added overhead causing WebTaint not being suitable for use in time- or memory sensitive domains. A use case for WebTaint is in test environments where security experts utilize the taint tracker to find TaintExceptions through manual or automatic attacks.

Bibliography

[1] "Android Stagefright vulnerability threatens all devices – and fixing it isn't that easy". eng. In: *Network Security* 2015.8 (Aug. 2015), pp. 1–2. ISSN: 1353-4858.

[2] S. Arzt et al. "FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps". In: vol. 49. 6. Association for Computing Machinery, June 2014, pp. 259–269.

[3] *ASM*. URL: http://asm.ow2.io/ (visited on 05/21/2018).

[4] Jennifer L Bayuk. *Cyber security policy guidebook*. eng. 2012. ISBN: 1-299-18932-6.

[5] J. Bell and G. Kaiser. "Phosphor: Illuminating dynamic data flow in commodity JVMs". In: *ACM SIGPLAN Notices* 49.10 (Dec. 2014), pp. 83–101. ISSN: 15232867.

[6] *Category:OWASP WebGoat Project - OWASP*. URL: https://www.owasp.org/index.php/Category:OWASP%7B%5C_%7DWebGoat%7B%5C_%7DProject (visited on 03/06/2018).

[7] "Chapter 1 - What is Information Security?" eng. In: *The Basics of Information Security*. 2014, pp. 1–22. ISBN: 978-0-12-800744-0.

[8] Justin Clarke-Salt. *SQL Injection Attacks and Defense, 2nd Edition*. eng. Syngress, June 2009. ISBN: 9781597499736.

[9] Iain D Craig. *Virtual Machines*. London : Springer London, 2006.

[10] Cristian Darie. *The Programmer's Guide to SQL*. eng. 2003. ISBN: 1-4302-0800-7.

[11] *Dynamic Security Taint Propagation in Java via Java Aspects*. URL: https://github.com/cdaller/security_taint_propagation (visited on 03/06/2018).

[12] William Enck et al. "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones". eng.

In: *Communications of the ACM* 57.3 (Mar. 2014), pp. 99–106. ISSN: 0001-0782.

[13] Seth Fogie. *XSS attacks cross-site scripting exploits and defense*. eng. Burlington, MA: Syngress, 2007. ISBN: 1-281-06024-0.

[14] *GitHub Octoverse 2017 | Highlights from the last twelve months*. URL: https://octoverse.github.com/%7B%5C#%7Dwork (visited on 03/21/2018).

[15] *Griffin Software Security Project*. URL: https://suif.stanford.edu/~livshits/work/griffin/ (visited on 05/21/2018).

[16] Vivek Haldar, Deepak Chandra, and Michael Franz. "Dynamic Taint Propagation for Java". In: (). URL: https://pdfs.semanticscholar.org/bf4a/9c25889069bb17e44332a87dc6e2651dce86.pdf.

[17] S.W. Hsiao et al. "PasDroid: Real-time security enhancement for android". In: *Proceedings - 2014 8th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2014*. Institute of Electrical and Electronics Engineers Inc., 2014, pp. 229–235. ISBN: 9781479943319.

[18] *HybridFlow*. URL: https://github.com/ylimit/HybridFlow (visited on 05/28/2018).

[19] *IBM Knowledge Center - -Xbootclasspath/p*. URL: https://www.ibm.com/support/knowledgecenter/en/SSYKE2%7B%5C_%7D8.0.0/com.ibm.java.lnx.80.doc/diag/appendixes/cmdline/Xbootclasspathp.html (visited on 03/20/2018).

[20] *Instrumentation (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html (visited on 02/27/2018).

[21] *Java virtual machine*. URL: https://en.wikipedia.org/wiki/Java_virtual_machine (visited on 06/04/2018).

[22] *java.lang.instrument (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html (visited on 05/21/2018).

[23] *Javassist by jboss-javassist*. URL: http://jboss-javassist.github.io/javassist/ (visited on 02/27/2018).

[24] *Locking Ruby in the Safe*. URL: http://ruby-doc.com/docs/ProgrammingRuby/html/taint.html (visited on 01/25/2018).

[25] Jianan Ma. "TaintDroid : An Information- - Flow Tracking System for Realtime Privacy Monitoring on Smartphones Jianan

Ma Problem Contribution / Implementation Details Questions / Suggestions". In: (2010), p. 3.

[26] *MAKE YOUR SOFTWARE SELF-PROTECTING*. URL: https://www.contrastsecurity.com/ (visited on 05/21/2018).

[27] Open Web Application Security Project. *OWASP*. URL: https://www.owasp.org/index.php/Main%7B%5C_%7DPage (visited on 02/01/2018).

[28] *OracleVoice: Java's 20 Years Of Innovation*. URL: https://www.forbes.com/sites/oracle/2015/05/20/javas-20-years-of-innovation/%7B%5C#%7D19a55ce611d7 (visited on 03/21/2018).

[29] OWASP. "OWASP Top 10 - The Ten Most Critical Web Application Security Risks". In: *Owasp* (2017), p. 22. URL: https://www.owasp.org/images/7/72/OWASP%7B%5C_%7DTop%7B%5C_%7D10-2017%7B%5C_%7D%7B%5C%%7D28en%7B%5C%%7D29.pdf.pdf%7B%5C%%7D0Ahttp://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:OWASP+Top+10+-2010%7B%5C#%7D1.

[30] *OWASP Insecure Web App Project*. URL: https://www.owasp.org/index.php/Category:OWASP_Insecure_Web_App_Project (visited on 05/16/2018).

[31] *OWASP Zed Attack Proxy Project*. URL: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project (visited on 05/17/2018).

[32] Jinkun Pan, Xiaoguang Mao, and Weishi Li. "Analyst-oriented taint analysis by taint path slicing and aggregation". In: *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS* 2015-November (2015), pp. 145–148. ISSN: 23270594. DOI: 10.1109/ICSESS.2015.7339024.

[33] *perlsec - perldoc.perl.org*. URL: http://perldoc.perl.org/perlsec.html (visited on 01/25/2018).

[34] *Phosphor: Dynamic Taint Tracking for the JVM*. URL: https://github.com/gmu-swe/phosphor (visited on 03/06/2018).

[35] *Primitive Data Types (The Java™ Tutorials > Learning the Java Language > Language Basics)*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (visited on 03/20/2018).

[36] *Python Taint*. URL: https://github.com/python-security/pyt (visited on 05/28/2018).

[37] *Same Origin Policy - Web Security*. URL: https://www.w3.org/Security/wiki/Same%7B%5C_%7DOrigin%7B%5C_%7DPolicy (visited on 02/07/2018).

[38] *Searching for Code in J2EE/Java*. URL: https://www.owasp.org/index.php/Searching_for_Code_in_J2EE/Java (visited on 05/21/2018).

[39] *SnipSnap - A java based wiki*. URL: https://github.com/thinkberg/snipsnap (visited on 05/16/2018).

[40] *Stanford SecuriBench Micro*. URL: https://suif.stanford.edu/%7B~%7Dlivshits/work/securibench-micro/ (visited on 03/15/2018).

[41] *Stanford SecuriBench Micro*. URL: https://suif.stanford.edu/~livshits/work/securibench-micro/intro.html (visited on 05/21/2018).

[42] Praveenkumat H Subbulakshmi T. *Secure Web Application Deployment Using Owasp Standards: An Expert Way of Secure Web Application Deployment*. Createspace Independent Publishing Platform, 2017.

[43] *The Aspectj Project*. URL: http://www.eclipse.org/aspectj/ (visited on 05/21/2018).

[44] *The benchmarks*. URL: http://dacapobench.org/benchmarks.html (visited on 05/16/2018).

[45] *The DaCapo Benchmark Suit*. URL: http://dacapobench.org/ (visited on 05/16/2018).

[46] *The Heartbleed Bug*. URL: http://heartbleed.com/ (visited on 05/28/2018).

[47] *The Java HotSpot Performance Engine Architecture*. URL: http://www.oracle.com/technetwork/java/whitepaper-135217.html (visited on 03/21/2018).

[48] *Ticketbook - This is a purposely insecure web application*. URL: https://github.com/Contrast-Security-OSS/ticketbook (visited on 05/17/2018).

[49] Guru Venkataramani et al. "FlexiTaint: A programmable accelerator for dynamic taint propagation". In: *Proceedings - International Symposium on High-Performance Computer Architecture* (2008), pp. 173–184. ISSN: 15300897. DOI: 10.1109/HPCA.2008.4658637.

[50] Bill Venners. *Inside the Java virtual machine*. McGraw-Hill, 1999.

[51] *What is Primo?* URL: https://kth-primo.hosted.exlibrisgroup.com/primo_library/libweb/action/search.do (visited on 06/04/2018).

[52] *Which methods should be considered "Sources", "Sinks" or "Sanitization" ?* URL: http://thecodemaster.net/methods-considered-sources-sinks-sanitization/ (visited on 05/21/2018).

[53] *Wikipedia - The Free Encyclopedia*. URL: https://www.wikipedia.org/ (visited on 05/21/2018).

[54] *World wide web skapas – nu kan internet bli en publiksuccé | Internetmuseum*. URL: https://www.internetmuseum.se/tidslinjen/www/ (visited on 03/06/2018).

Appendix A

Raw Data

Appendix A consist of two tables containing raw data possibly not included in the thesis. The Tables A.1 and A.2 contains average, min and max values from conducting the micro-benchmarks evaluation.

Table A.1: Time measurements (ms) from executing The DaCapo Benchmark Suite, with and without WebTaint, ten times.

	No Taint Tracker			WebTaint		
	Average	**Min**	**Max**	**Average**	**Min**	**Max**
Avrora	3813025	3744824	3866363	9042154	8325428	9523650
Batik	2643695	2351608	3837237	14068644	12514609	17751412
Eclipse	38284019	35090309	40662754	53031768	49999425	55297291
Fop	2100317	1976965	2264453	11050875	9449910	11701099
H2	14879971	14285215	15269910	24409953	23402474	25453261
Jython	10867700	10323676	11154908	26884920	26013407	29497966
Luindex	1753020	1662680	1838984	4860207	4402878	5456444
Lusearch	2902191	2691449	3184846	5957591	5529709	6498355
Pmd	3103044	2978561	3319209	10713312	10198144	11478354
Sunflow	5145955	4967500	5396681	11039976	10644328	11523814
Tomcat	7871662	7654701	8316705	21592218	19901562	22886977
Tradebeans	113344823	15936751	124316871	143159947	142096360	144361149
Tradesoap	124208601	124032117	124326210	142446607	141075967	144368091
Xalan	3742703	3493600	4132797	10366234	9518026	11132662

Table A.2: Memory measurements (kilobytes) from executing The DaCapo Benchmark Suite, with and without WebTaint, ten times.

	No Taint Tracker			WebTaint		
	Average	**Min**	**Max**	**Average**	**Min**	**Max**
Avrora	108445	99716	122236	336260	240968	407668
Batik	178804	173812	185520	794894	659808	863608
Eclipse	922929	916340	938032	973240	954060	1024412
Fop	167038	141788	207216	631080	507636	810200
H2	842447	802652	865792	979604	967580	1000056
Jython	730460	620336	764108	862572	846948	880192
Luindex	102332	97736	105760	285066	226780	316556
Lusearch	276592	213280	333340	464162	343036	621868
Pmd	246932	232384	272068	546636	442624	700996
Sunflow	333194	311008	466532	722237	640484	796664
Tomcat	392682	315292	442928	847140	690324	898144
Tradebeans	371280	281796	688620	926053	916492	938524
Tradesoap	307335	278072	380244	919946	896588	935964
Xalan	235313	180188	362980	650827	563332	670492