WebTaint: Dynamic Taint Tracking for Java-based Web Applications

FREDRIK ADOLFSSON

Master in Computer Science
Date: June 20, 2018
Supervisor: Musard Balliu
Examiner: Mads Dam
Swedish title: WebTaint: Dynamic Taint Tracking för Java-baserade webbapplikationer
School of Computer Science and Communication

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, to communicate with customers, and to create new business opportunities. However, this does not come without drawbacks as there exists an elevated risk to become targeted in attacks.

The thesis implemented a dynamic taint tracker, named 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 possible advantages and disadvantages by using the application is introduced as well as an explication whether the application was capable of being integrated into production services.

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. WebTaint could be recommended for use in test environments where security experts utilize the taint tracker to find TaintExceptions through manual and 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 emellertid inte utan nackdelar, eftersom det finns en förhöjd risk att bli måltavlor i attacker.

I avhandlingen implementerades en dynamic taint tracker, namngett WebTaint, med uppgift att förhindra sekretess och integritetsproblem i Java-baserade webbapplikationer. Vi utvärderade i vilken utsträckning WebTaint kan bekämpa integritets sårbarheter. De möjliga fördelarna och nackdelarna med användning av applikationen introduceras såväl som en förklaring om applikationen kunde integreras i produktionstjänster.

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 och automatiska attacker.

Contents

List of Tables

List of Figures

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 essential source of information and it connects the world socially through a single platform. Furthermore, many businesses have taken advantage of the World Wide Web to share information, to communicate with customers, and to 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 causes harm to other users and companies behind the web applications.

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

All applications with accessibility from the web share the same 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. Therefore untrusted data needs to be sanitized before being used. The consequences of using untrusted data could be catastrophic. A variety of tools have therefore been created to minimize the risk of accidentally introducing security flaws into web applications. One of these tools is the taint tracker, which attempts to secure the applications through information flow control. The taint tracker 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 up the questions how useful the taint tracker is for web applications and also 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?*

When developing web applications it is recognized that safety is a growing problem and that work towards protecting user data is necessary. 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 procedure to fight the problem with unsanitized user input has a diversity of solutions where taint tracking is one. The purpose is to protect the web applications by implementing a dynamic taint tracker which both runs and analyzes the code at runtime. However, the question to ask is to what extent the taint tracker will protect the application. Also, what advantages and disadvantages it will entail. Furthermore, if the solution is capable of being integrated into production services.

1.2 Aim

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

The implementation of WebTaint aims to allow tracking of taint for Strings including all data types used for String operations. These data types are String, StringBuilder, StringBuffer, CharArray and ByteArray. The evaluation of WebTaint will be conducted through case studies and micro-benchmarks to measure the detection rate of vulnerabilities and introduced performance overhead. Concretely, we will implement and evaluate a WebTaint against SQL Injection and Cross-Site Scripting vulnerabilities.

1.3 Contribution

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

- Implementation of WebTaint. A dynamic taint tracker for Java-based web applications.
- Evaluation of WebTaint for vulnerability detection rate and introduced performance overhead.
- Discussing and drawing conclusions regarding the use of dynamic taint trackers for Java-based web applications.

1.4 Limitations

The focus of the thesis lies in the security vulnerabilities of web applications. However, other application areas might be subjected to the same kind of vulnerabilities. This thesis will neither discuss or present information regarding those areas to keep the scope of the thesis at a reasonable level. Another limitation to keep the reasonable level of the scope is the decision of not discussing or presenting all types of Injection vulnerabilities. Injection attacks are a broad vulnerability, but this thesis will focus on Injection attacks towards SQL.

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 introduced overhead and detection of integrity vulnerabilities. This is due to time limitations.

1.5 Methodology

To answer how taint tracking can secure Java-based web applications, we use a combination of both 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, on the other hand, consists of the implementation and evaluation of WebTaint, where benchmarks for introduced overhead and security assurance will be performed. The introduced overhead is conducted with the DaCapo Benchmark Suite [45] consisting of fourteen applications mimicking common Java applications. Security assurance is evaluated through case studies where four applications are tested.

1.6 Ethics & Sustainability

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

So, the use of WebTaint result in more robust and secure applications thanks to the ability to find possible security vulnerabilities and giving the ability to fix them. WebTaint could be used by the developers in the daily work to validate the soundness of the implemented code. This would ease some of the work for the developer.

As far as we can see there is only one unethical aspect of WebTaint and that is the fact that the taint tracker gains access to all data processed by the application. Therefore, a proper implementation of the taint tracker is essential to ensure that it is not possible to abuse the taint tracker to gain access to information.

1.7 Outline

The thesis outline starts with the *Background* presenting information regarding the subject. Next follows *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 introduced overhead results. Lastly comes the *Discussion*, *Future Work* and *Conclusion* chapters.

Chapter 2

Background

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

2.1 Web Application

To make applications available and accessible from almost everywhere, 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 part and contains the applications business logic. The third tier is the storage tier, where the business logic stores data as needed [8].

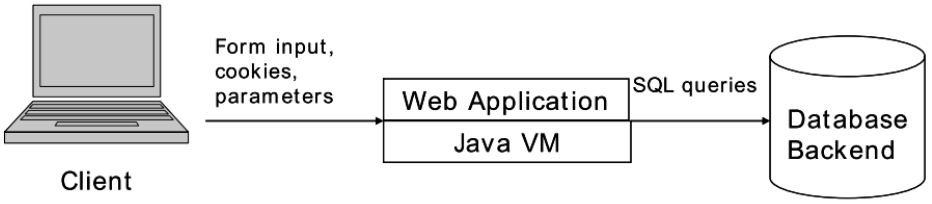

Figure 2.1: An illustration of the three-tier architecture commonly used by web applications [17].

From Figure 2.1 it can be seen that a tier only communicates with the tier closest to themselves. This demands 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, consist of username, 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. This is the tier where validation is needed to ensure security. The programming language for the logic tier can vary a lot, but one commonly used 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. The SQL is created to manipulate and access databases programmatically. The majority of today's database uses SQL [11]. The language works by building queries specifying the required information or task. The query is then evaluated and handled by the SQL engine [10].

2.2 CIA Triad

Discussions regarding information security often rely on the CIA Triad. The CIA refers to confidentiality, integrity, and availability as displayed in Figure 2.2. Confidentiality ensures that data is only accessed by authorized individuals. Integrity specifies that application data should be accurate and unaltered. While availability is the ability to access the application and application data [7].

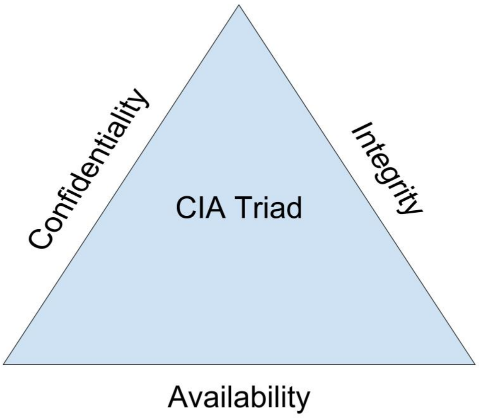

Figure 2.2: An illustration of the CIA Triad, model used when discussing information security.

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]. The 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 the possible prevention methods are presented. This thesis will focus on security risk number one and seven from the mentioned report. These two security risks deal with vulnerabilities regarding 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]. An 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 are executed towards a broad set of different areas, but the area discussed and analyzed in this thesis are SQL Injections. The SQL Injections can be divided into two different subgroups. These two subgroups are SQL Injection and Blind SQL Injection [42].

SQL Injection

The SQL Injection occurs when an SQL query is tampered with, resulting 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 due to that the variable UserId never is 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 works 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 and therefore returns the whole table of users. This problem can be prevented in a couple of different ways. The first possibility is through validation of input, by verifying user input as in Listing 2.3 it is possible to protect the query from being vulnerable to a 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 to resolve the attack 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

There also exists a blind SQL Injection which is very similar to the 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 kind 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, 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

Another security vulnerability is Cross-Site Scripting which 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 wanted to gain access to. The attacker could then, through JavaScript, access any content visible or typed into the web application. The Same-Origin Policy was therefore introduced to prevent this form of attacks. The policy restricts JavaScript to only access content from its origin [14, 36].

The introduction of the Same-Origin Policy, however, did not stop the attackers. The next wave of attacks was mostly directed towards chat rooms where it was possible to inject malicious Cross-Site Scripts into the message input form. This would then be reflected by the server itself, when displaying the message for other users, and thereby bypassing the Same-Origin Policy [14].

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 is mainly 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 means that 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

We have now presented a set of problems that the Taint tracking, also known as taint analysis, is a tool to combat. The tool analyzes the flow of information in the application [32]. The goal of taint tracking is to prevent possible attacks such as the Injection and Cross-Site Scripting by enforcing the usage of sanitizers on every input data. The Taint tracking can be implemented in two different forms: either static or dynamic. The static taint tracking is an evaluation tool possible to include in the integrated development environment where it notifies the developer of possible security vulnerabilities. The dynamic taint tracking, on the other hand, is a tool used simultaneously as the application execution. The Dynamic tracking analyses the input data to discover vulnerabilities at runtime and achieve higher accuracy compared to static tracking. The advantage with the static form is the ability to run before runtime, but its disadvantage is the lower accuracy in tracking of taint.

The Taint trackers operate by tracking untrusted data and acting upon anything that is trying to enter sensitive code areas without first being sanitized. Perl and Ruby are two programming languages which have been adapted to use taint checking [33, 24]. There are some tools which enable taint checking for other platforms. One of them is TaintDroid [25] for the Android platform.

The process of taint tracking consists of four steps which are described in Table 2.1. The first step is to mark all data from untrusted sources as tainted. This is done through a taint flag attached to the input variables. Step two is the possibility of detainting data, but this is only done after that the data has been sanitized through predefined sanitizers. The third step is propagating taint where tainted data propagates its tainted flag onto all data it comes in contact with. 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 owner´s 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.
Detainting	Marking all data from sanitizers as non-tainted.
Taint Propagation	Propagating taint to all data coming in contact with tainted data.
Assert Non-taint	Assert that data passing through sinks are non-tainted.

An example of the taint propagation process can be seen in Listing 2.6. In the example *getAttribute* is a source, *executeQuery* is a sink and *validate* is a sanitizer. On line one, the input from the source is flagged as tainted, and the taint propagates onto *userId*. The sanitizer on line two validates *userId* and removes the tainted flag. Lastly, the sink on line tree 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, on the other hand, 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.

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

2.5 Java

Java has been a programming language in use 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]. Still today Java is one of the most common programming languages [15].

Java is a statically typed language which means that no variable can be in use before being declared. The variables can be of two different types: either primitives or as references to objects. Among the primitive types does Java have support for the eight following: 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 the HotSpot [47]. One of the core ideas with Java during its development was to "write once, run anywhere." The slogan was created by Sun Microsystems which at the time was 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 bytecode 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].

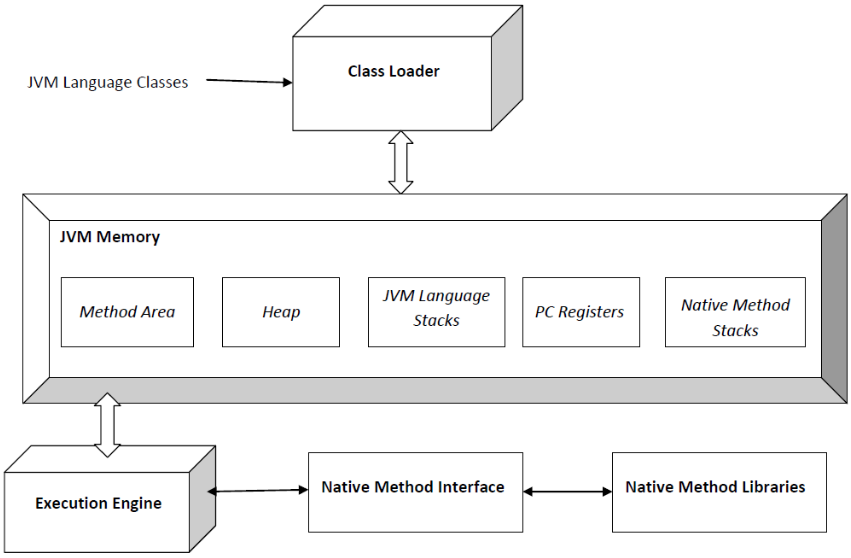

Figure 2.3: An illustration of the Java Virtual Machine Architecture [21].

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. The 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 be of help to 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*. These 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 within the field. It addresses the areas of dynamic and static taint tracking for Java, as well as other related platforms.

Haldar et al. [17] has written a report about dynamic taint tracking for Java applications where the authors tried to solve the problem of correct user input validation. They built a tool that is independent of the web applications source code and the results from using the tool were proven to prevent code injection. Haldar et al. [17] evaluated the taint tracker on OWASP WebGoat [6] but acknowledged that benchmarks of real-world web applications are needed to validate the implemented tools functionality. Their application supports taint tracking for the String class by adding a taint flag and altered the methods to propagate the taint. The tool Haldar et al. [17] implementation cannot be found for use for further evaluations.

Another implementation of a dynamic taint tracker for Java applications is Phosphor [34] created by Bell and Kaiser [5]. Phosphor was developed with the help of the Java bytecode manipulation library ASM [3]. The application addresses taint tracking for primitives and arrays by introducing shadow variables. A shadow variable is a variable holding the taint flag for an un-instrumentable object. The shadow variable is injected 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-instrumented object. Phosphor does however not support detainting of variables [5] which makes the application not suitable for finding code injection vulnerabilities.

A third implementation of a dynamic taint tracker for Java applications is the Dynamic Security Taint Propagation [12] 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 tool relies on aspect-oriented events that trigger the taint propagation, tainting data from sources, and assertions that ensure that tainted values do not pass through sinks. The application does however not support the ability to add or remove sources, sinks, and sanitizers. This lack of feature causes that the usability of the application becomes restricted to the current implementation.

In addition to dynamic taint trackers for Java applications, there exist dynamic taint trackers for other platforms. One of these is TaintDroid which is constructed for Android smartphones and aims to prevent privacy violations [13]. TaintDroid uses a shadow memory which reduces the memory overhead of the application. Hsiao et al. [18] has as well conducted further work on top of TaintDroid where is created a security scheme called PasDroid. PasDroid enables users to gain full control over the management of possible information leaks on Android devices.

There exist several static taint trackers, e.g., FlowDroid by Arzt et al. [2]. FlowDroid computes data flows for Java and Android applications. The static tracker is built with the help of the Soot framework [39] providing analysis tools used to find possible vulnerabilities. The FlowDroid is evaluated in comparison to The AppScan and Fortify. The results show that the FlowDroid has a higher true positive and lower false positive rates.

Chapter 4

Implementation

This chapter presents the implementation process of WebTaint. Chapter starts with a section describing Policies enforced by the application. Thereafter a description of how Sources, Sinks & Sanitizers were specified and lastly an explanation of the WebTaint architecture.

4.1 Policies

To be able to implement the functionality behind the taint tracker, security policies need to be defined. The security policies imply the principles and actions that the application strives to fulfill [4]. The taint tracker developed in this thesis aims to fulfill two different kinds of policies. These are *integrity* and *confidentiality*.

4.1.1 Integrity

The first policy is the integrity policy which is important as it defines that users may not modify data which they do not have permission to alter. The integrity policy aims to protect from, e.g., injections of malicious code that can lead to information disclosure or destruction of user data. To ensure this, we define the following policy:

- Users shall alter no data or execution without having the correct permission for the given data or execution.

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

4.1.2 Confidentiality

The second policy is the confidentiality policy which defines that data given to the user should only be data that the user have the right to access. This goal concern the prevention of malicious usage where attackers wish to steal application data. To ensure this, we define the following policy:

- Users shall not gain access to data without having the correct permission for the given data.

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

While monitoring for confidentiality vulnerabilities are the sources, and sinks roles reversed compared to integrity flow analysis. The information wanted to protect is marked as sources and sinks are the systems public exit points.

4.1.3 WebTaint

The two sections above state what both the integrity and the confidentiality policies of WebTaints aim to fulfill. The two policies can also be rewritten into policies for WebTaints internal logic. The internal policies are divided into the four tasks described in Table 2.1 in Chapter 2. These tasks are tainting, detainting, taint propagation, and assertion of non-tainted data.

Tainting

Table 4.1 presents the tainting policies of WebTaint.

Table 4.1: WebTaint's tainting policies.

Integrity	Data from untrusted sources shall always be marked tainted.
Confidentiality	Data from private sources shall always be marked tainted.

Detainting

Table 4.2 presents the detainting policy of WebTaint.

Table 4.2: WebTaint's detainting policy.

Integrity & Confidentiality	Data from sanitizers shall be marked as detainted.

Taint Propagation

Table 4.3 presents the taint propagation policy of WebTaint.

Table 4.3: WebTaint's taint propagation policy.

Integrity & Confidentiality	Data resulting from tainted data shall be marked tainted.

Assertion of Non-taint

Table 4.4 presents the assertion of non-tainted data policies of WebTaint.

Table 4.4: WebTaint's assertion of non-taint policies.

Integrity	No untrusted data may pass through a trusted sink.
Confidentiality	No private data may pass through a public sink.

4.2 Sources, Sinks & Sanitizers

Defining the sources, sinks, and sanitizers is a large task in itself. There is no official documentation in Java specifying these for web applications. Depending on what application, framework, and library used the sources, sinks, and sanitizers will vary a lot. The sources, sinks, and sanitizers used in this thesis is, however, an aggregation from *Which methods should be considered "Sources", "Sinks" or "Sanitization" ?* [52] and *Searching for Code in J2EE/Java* [37]. The web pages contain lists of sources, sinks, and sanitizers based on the author's experience in web application security. These lists are aggregated into one JSON file per each source, sink, and sanitizer containing classes and method names to instrument.

4.3 WebTaint

The implementation of WebTaint was divided into three subprojects. The reasoning behind the separation was because of the need of transforming classes both before and during runtime as presented in Section 2.5.2. These three projects are called Agent, Xboot, and Utils. The Agent handles the transformation in runtime and Xboot transforms the classes in the Base Java Runtime Environment. The logic of class transformation is the same in both projects and therefore centralized in the Utils subproject. A short description of the three subprojects is presented in Table 4.5.

Table 4.5: Description of the three subprojects in WebTaint.

Agent	Running at runtime and triggering the transformation of sources, sinks, and sanitizers.
Xboot	Running prior to runtime and transforming sources, sinks, sanitizers, String, StringBuilder, and StringBuffer in the Base Java Runtime Environment.
Utils	Utilities to transform classes into sources, sinks, and sanitizers.

Transformation of the String, StringBuilder, and StringBuffer classes is done in the Xboot project. This is done by transforming the classes to:

- Contain a boolean to hold the taint flag.
- Propagate taint in each method and constructor (taint propagation policy 4.3).

A high-level architecture of WebTaint instrumenting a web application is seen in Figure 4.1. The Java Virtual Machine is un-instrumented, and so is WebTaint. The web application and all its depending libraries, however, are instrumented.

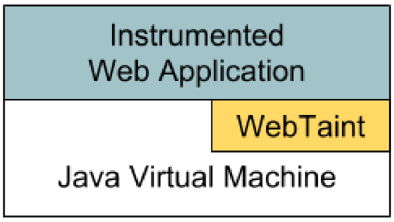

Figure 4.1: High-level architecture of WebTaint running on the JVM.

Out of the three subprojects, it is the Utils project that contains WebTaint's core logic. The Agent and Xboot act as a pipeline providing the Utils project with classes to analyze and possibly instrument. Therefore is only the logic behind the Utils project presented in the chapter below.

4.3.1 The Utils Project

The Utils project includes the core logic of marking methods as sources, sinks, and sanitizers. The significant part of the implementation is the same for the three different types, and the only difference is how they get instrumented. The logic behind finding the classes to instrument work by taking all classes and matching them with the three criteria in Table 4.6. This makes it possible to find if the class is a defined or builds upon a defined source, sink or sanitizer.

Table 4.6: Description of the three criteria of finding sources, sinks and sanitizers to instrument.

Is Class	The class is listed as a source, sink or sanitizer.
Implements Interface	The class implements interface of a listed source, sink or sanitizer.
Extends Class	The class extends a class which [Is Class], [Implements Interface] or [Extends Class].

Classes found to be a source, sinks or sanitizer are instrumented by iterating through the class methods and comparing them with the methods defined in the JSON file corresponding to the matched source, sink or sanitizer. These matched methods will be instrumented depending on the type. What instrumentation is done per type is seen in Table 4.7.

Table 4.7: Description of what logic is instrumented into the application depending on if the class method is a source, sink or sanitizer.

Source	Tainting the current and return object (tainting policies 4.1).
Sink	Asserting the current object and method arguments to be non-tainted (detainting policy 4.4)
Sanitizer	Deainting the current and return object (detainting policy 4.2).

The instrumented logic of sinks is an assertion of the current object and arguments to not be tainted. If this assertion comes back incorrect, a remedial action is needed. This action could be, a logging event, throwing a TaintException or modifying the tainted value into a safe predefined value. During the evaluations, the option of using a predefined value and logging the event is used.

4.3.2 Limitations

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, the solution to this 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* [19] that appends the classes in the front of the bootstrap classpath. This triggers the Java Virtual Machine to utilize the instrumented rt.jar before the original version.

Another problem is that instrumentation of primitives and arrays is not possible. The implementation of WebTaint aims to support propagation of taint for Strings. However, the problem of not being able to instrument primitives and arrays creates the risk of possibly losing the taint when string operations are done with the help of byteArrays or charArrays. So, the solution to this is to create shadow variables as Bell and Kaiser [5] did while developing Phosphor [34]. However, another possible solution is to create a centralized memory bank just as Enck et al. [13] did when implementing TaintDroid.

There is also the problem with primitive operations which 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. Hence, solving these operations on bytecode level is therefore necessary. [23].

Chapter 5

Evaluation

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

5.1 Test Environment

The benchmarks are conducted on an Asus Zenbook UZ32LN. 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

To evaluate the usefulness of WebTaint benchmarking of two kinds are conducted. The first are case studies on web applications where the increase in security by using WebTaint is measured. The second kind of benchmarks are a set of micro-benchmarks where the introduced overhead is measured.

Every evaluation for both of the benchmarks are conducted twice. The first time is without WebTaint and the second is with theWebTaint. The reason for this is to acquire the baseline values and the values when using WebTaint. The difference between these is of interest because they will indicate the security increase and introduced overhead.

5.2.1 Web Applications

WebTaint does not detect vulnerabilities in applications by itself and needs external actions to trigger executions. These executions can WebTaint then analyze for vulnerabilities. To trigger executions in the applications, we used OWASP Zed Attack Proxy [31], known as ZAP. The ZAP is an open-source testing tool used to scan web applications for security vulnerabilities and is widely used in the penetration testing industry. A high-level illustration of usage of ZAP can be seen in Figure 5.1.

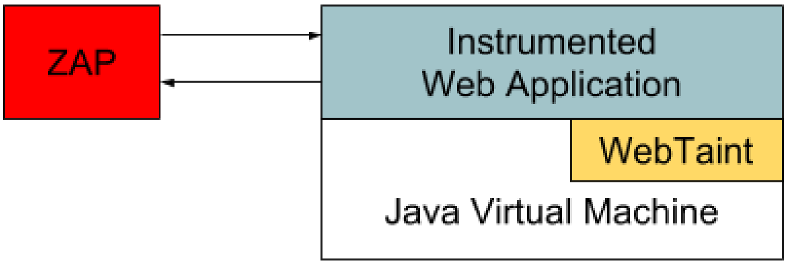

Figure 5.1: High-level architecture of the ZAP analysing WebTaint enabled web application.

The ZAP accesses the web application over the HTTP protocol and searches for possible vulnerabilities in the applications. An advantage of utilizing a penetration testing tool for evaluation of WebTaint is that vulnerabilities detected by WebTaint are assured to be vulnerabilities if the ZAP detects them as well. However, WebTaint has the possibility of detecting a more extensive set of vulnerabilities since it analyzes applications internally compared to the ZAP which only analyzes externally.

The evaluations of applications are a time-consuming task and therefore only conducted on a smaller set of applications. Each application is a Java-based web application vulnerable to the SQL Injections and Cross-Site Scripting attacks. The web applications are presented in the sections below.

Stanford SecuriBench Micro

The Stanford SecuriBench Micro is a set of small test cases designed to evaluate security analyzers. The test suit is deliberately insecure and was created as part of the Griffin Security Project [16] at Stanford University. The application contains 96 test cases and is written in 46407 lines of code. The tests are accessible through input fields on the application. This thesis uses version 1.08 of the application, which is known to contain the vulnerabilities the SQL Injection, Cross-Site Scripting, HTTP Splitting, Path Traversal and more [40, 41].

InsecureWebApp

The InsecureWebApp is a deliberately insecure web application developed by OWASP to detect security vulnerabilities and possible harm caused by them. The web application is built for a fictional company called American Service Corporation. Some of the functionalities the application provides are registering, log in, product browsing and placing money into the company account. The vulnerabilities are accessible through the applications input fields and HTTP parameters, and some of them are the Parameter Tampering, Broken Authentication, SQL Injection, HTML Injection, Cross-Site Scripting. The project consists of 2913 lines of code and version 1.0 is used in this thesis [30].

Ticketbook

The Ticketbook is a deliberately insecure web application developed by Contrast Security to show the power of one of their security tools.

The application consists of a set of pages illustrating different security vulnerabilities accessible through input fields and HTTP parameters. Some of these vulnerabilities are the Cross-Site Scripting, Command Injection, SQL Injection, Parameter Tampering, XML External Entity. The application consists of 13849 lines of code and version 0.9.1-SNAPSHOT is used in this thesis [48, 26].

SnipSnap

The SnipSnap is developed to provide the necessary infrastructure to create a collaborative encyclopedia. The web page functionality is similar to the Wikipedia [53] where users can sign up and contribute by writing posts. The vulnerabilities in the application are accessible through the different input fields the application utilizes to provide functionalities such as registering and log in. The SnipSnap consists of 566173 lines of code and version 1.0-BETA-1 is used in this thesis [38]. The application is not constructed to be deliberately insecure and is intended to be used in production.

Web Application Policies

The web applications presented in the previous sections all strive to provide services fulfilling the same policies as described in Section 4.1.1 and 4.1.2. This emphasizes the fact that all the input fields, HTTP parameters, and other possible user modifiable data are sanitized before being used in the web applications. Possible harmful scenarios of using un-sanitized user input are for example database actions and reflecting information to the user.

5.2.2 Micro Benchmarks

The introduced overhead is measured by the added time and memory overhead. To evaluate these two the DaCapo Benchmark Suit [45] is used. The DaCapo is a set of applications constructed specifically for benchmarking of Java applications. This thesis uses version DaCapo-9.12-bach which consists of fourteen real-world applications. Table 5.1

contains a description for each application. The summary is taken from the DaCapos website [44].

Table 5.1: Descriptions for each application in the DaCapo Benchmark Suit [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 introduced time and memory is conducted through a C script constructed to execute each application in the DaCapo Benchmark Suit ten times, both with and without WebTaint. Each measurement is executed in an isolated process one at a time to allow accurate memory and time measurements.

Chapter 6

Result

This chapter presents the evaluation results. Appendix A contains complemented data and metrics which will not be shown in this chapter. Chapter starts with exposing the results of the Web Applications evaluation. This follows by results presented from the Introduced Overhead evaluation.

6.1 Web Applications

The results presented in this section are from evaluating Java applications for security vulnerabilities with and without WebTaint. The results from each application are listed in tables where vulnerability type and the vulnerability count are presented.

Table 6.1 shows the vulnerabilities from evaluating Stanford SecuriBench Micro [40]. In the table we can see that the most common vulnerability is Reflected Cross-Site Scripting where 71 vulnerabilities are presented. Second most common is SQL Injection with 20 and the least common with one vulnerability is Buffer Overflow. By enabling WebTaint 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	Detected by WebTaint
Cross-Site Scripting (Reflected)	71	71
SQL Injection	20	20
Buffer Overflow	1	1

Table 6.2 shows the vulnerabilities from running the InsecureWebApp [30] with and without WebTaint. Of the two types of vulnerabilities is SQL Injection the most common with six vulnerabilities and Reflected Cross-Site Scripting with two. Enabling WebTaint on the 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	Detected 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 the Ticketbook [48]. The most common vulnerability was the Cross-Site Scripting with 14 occurrences. The SQL Injection was the least with one. The prevention rate of the SQL Injection was 100% and for Cross-Site Scripting 71.4% when enabling WebTaint. The overall prevention rate is 73.3%.

Table 6.3: Security vulnerabilities detected by WebTaint in Ticketbook

	Vulnerabilities	Detected 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 [38] is seen in Table 6.4. In this table can we see that the most common vulnerability is the Reflected Cross-Site Scripting with 172 occurrences. Second highest is the SQL Injection with 49 occurrences followed by the CRLF Injection with two. Enabling WebTaint yields an overall prevention rate of 77.2%. All CRLF Injection is prevented. The Cross-Site Scripting prevented with 77.3% and the SQL Injection with 75.5%.

Table 6.4: Security vulnerabilities detected by WebTaint in SnipSnap

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

The results indicate that detection and prevention of the SQL Injection and the Criss-Site Scripting attacks are possible. Detection of vulnerabilities is possible for all applications and the average prevention rate for all four applications is 81%.

6.2 Introduced Overhead

The results from benchmarking the application on the 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 WebTaint enabled. The graphs are constructed based on the data in Table A.1 and A.2 in appendix A.

6.2.1 Time

Figure 6.1 displays the results of the average time overhead per application when enabling WebTaint. The results show that the application with the least average time overhead was the Tradesoap where 14.7%

was added. The application with the most average time overhead was the Batik with an overhead of 432.2%. The average overall is 162.9%.

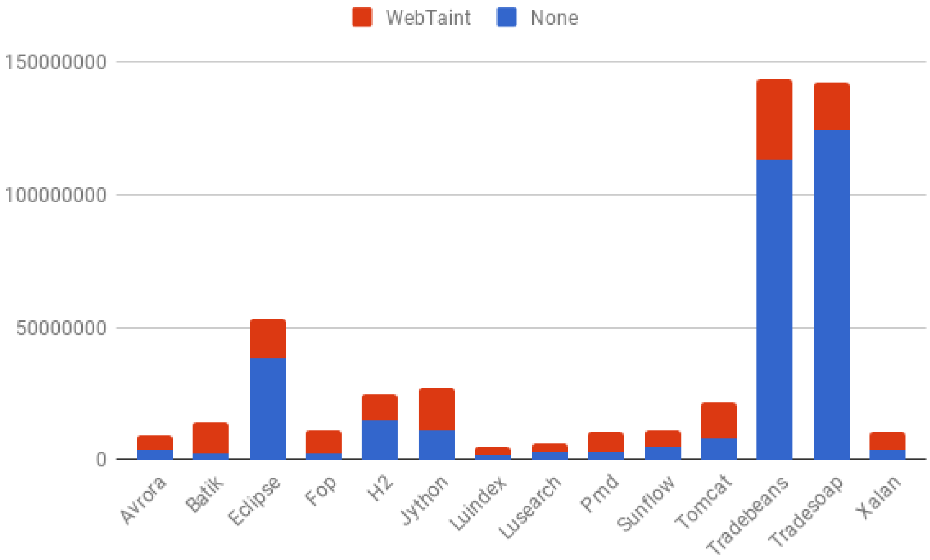

Figure 6.1: Average added time in microseconds

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 the Eclipse where 5.5% was added. The largest application was the Batik with an overhead of 344.6%. The average overall is 142.7%.

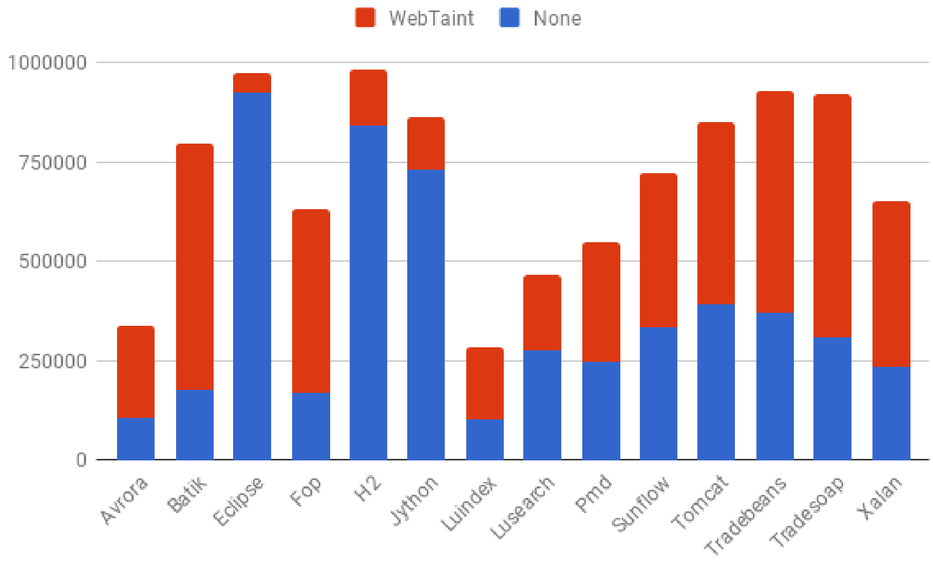

Figure 6.2: Average added memory in kilobytes

Chapter 7

Discussion

This chapter contains discussions regarding the implemented dynamic taint tracker, named WebTaint, and how well it performs. Chapter starts with a general discussion. This is then followed by Taint Propagation discussion. The last two sections are discussions about Sources, Sinks & Sanitizers and the Methodology of Evaluation.

By looking at the results in the previous chapter we see a clear indication of that WebTaint is capable of detecting security vulnerabilities. The Stanford SecuriBench Micro has 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%. This indicates a significant impact in combating integrity vulnerabilities. The prevention rate could also be increased by the further development of the application where taint tracking support is enabled for charArrays and byteArrays, which however were not implemented due to time limitations during this thesis.

Despite, the increase in security might not in the end be worth it if significant drawbacks are implicated. From the overhead results can we see the use of WebTaint introduces overhead. This overhead comes from the Java Agent instrumenting the classes and the added operations to propagate taint. The application domains where time and memory usage is not a problem would therefore not suffer from the introduced overhead. However, web applications need fast response times to provide a good user experience for its users. This causes them to be time sensitive and that is a reason why WebTaint is not suitable to be included in production systems.

The most significant impact on time overhead comes from the startup phase of the application where the Java Agent instruments classes files. Instrumentation of a class file happens only once, and it is done the first time the Class Loader loads the file. This means that applications executed for an extended period and reuses a smaller set of class files are less affected by the time overhead. It is shown in Figure 6.1 where Avrora's and Batika's time overhead are 137.2% respectively 432.2% compared to Tradebeans and Tradesoaps 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, the H2 and the Jython DaCapo tests that have significantly fewer numbers. It is hard to interpret why these are significantly less. One guess would be that they significantly use a smaller amount of strings and therefore are not affected by the added taint flag and other help functions instrumented into the String, StringBuilder, and StringBuffer classes.

7.1 Taint Propagation

Due to time issues, only the classes String, StringBuilder and StringBuffer were implemented to support taint propagation. The limitation is justifiable as these are the most important classes for taint tracking when securing web applications. However, there is important to take into account the risk of losing the tracking of taint since some libraries use charArrays and byteArrays for String operations. The results prove that the implemented classes have a significant influence on the outcome. Nevertheless, the optimal solution would be with complete integration for all data types in Java. Just like the Phosphor, but with the ability to sanitize variables.

7.2 Sources, Sinks & Sanitizers

During the planning phase of the thesis was the task of defining sources, sinks and sanitizers believed to be minor task. Taint trackers depend on these and it is essential to dedicate work to define these correctly. However, this was a time-consuming task that was out of the scope of the thesis. The solution was to compile definitions of sources, sinks, and sanitizers from sources found online. These consisted of bloggers putting together what they believed some of the sources, sinks, and sanitizers should be.

The optimal solution to define sources, sinks and sanitizers would be extensive research where lists for each library, framework, and deployment utility used by Java-based web applications were compiled. These lists could then be used depending on what functionality the implemented web application is using. The best situation 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. Multiple taint types would lead to a more advanced taint tracking where data from a specific type of sources are sanitized with the correct type of sanitizer. This would remove the risk of mistakenly using incorrect sanitizers and also ensure better protection of WebTaint.

7.3 Methodology of Evaluation

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

Chapter 8

Future Work

There is some work needed to be done before WebTaint can take place as an adequate solution to secure web applications from security vulnerabilities. One thing necessary to do is to finalize the 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. Implementing different types would allow for an advanced taint tracking where sanitizers only capable of sanitizing one specific type of data and do not mark other types as safe.

Optimizations of WebTaint's execution time and memory usage is also in need since it was not prioritized during this thesis. Improvements minimizing the introduced overhead would make WebTaint useful 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 currently supported by WebTaint, are charArrays and byteArrays.

Another possible WebTaint enhancement is to implement support of implicit taint flows. WebTaint does at the moment only support explicit flows where taints propagate if the calculated variables are directly dependent on a tainted variable. For example, x in $x = y + 1$ would 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 in the thesis a dynamic taint tracker for Java-based web applications named WebTaint. The goal was that the tool could combat integrity and confidentiality vulnerabilities. The results of the conducted evaluations show improved security when using WebTaint. However, there are drawbacks regarding overhead causing WebTaint not presently being suitable for use in time- or memory sensitive domains. WebTaint could be recommended for use in test environments where security experts utilize the taint tracker to find TaintExceptions through manual and 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] *Developer Survey Results 2018*. URL: https://insights.stackoverflow.com/survey/2018/ (visited on 06/20/2018).

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

[13] 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.

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

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

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

[17] Vivek Haldar et al. "Dynamic Taint Propagation for Java". In: (). URL: `https://pdfs.semanticscholar.org/bf4a/9c25889069bb17e44332a87dc6e2651dce86.pdf`.

[18] 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.

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

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

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

[39] *Soot - A Java optimization framework*. URL: https://github.com/Sable/soot.

[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-benchmark evaluations.

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