WebTaint: Dynamic Taint Tracking for Java-based Web Applications (DRAFT)

FREDRIK ADOLFSSON

Master in Computer Science
Date: May 30, 2018
Supervisor: Musard Balliu
Examiner: Mads Dam
Swedish title: WebTaint: Dynamic Taint Tracking för Java-baserade webbapplikationer
School of Computer Science and Communication

Abstract

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

The internet is a source of information, and it connects the world through a single platform. Many businesses have decided to take advantage of the web platform to share information and communicate with customers. However, this does not come without drawbacks. There are several potential attacks can cause harm to a web application. The attack most frequently conducted today will probably not be the same as the most performed in the future. 2017 was Injection Attacks and Cross-Site Scripting among the ten most frequently conducted attacks.

This thesis implements and evaluates a dynamic taint tracker to prevent confidentiality and integrity vulnerabilities in Java-based web applications. Does dynamic taint tracking enforce the same security gains as Domain-Driven Security?

The results show that dynamic taint tracking helps to combat Injection and Cross-Site Scripting attacks just as Domain-Driven Security. However, there are drawbacks in the form of additional time and memory overhead. Which in this case is quite significant.

Sammanfattning

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

Contents

Chapter 1

Introduction

The creation of the World Wide Web has caused a significant impact on today's society [50]. 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. Software as a Service is one of these opportunities where companies provide software over the internet. [1, 16, 21] However, this advancement in technology does not come without its drawbacks. The web applications are not only accessible to the targeted user groups but anyone with access to the web. This entails that malicious users who wish to abuse and cause harm to other users have the accessibility to do so possibly.

There have been a plethora of incidents causing vulnerabilities resulting in everything from money loss, disclosure or destruction of information. One of these 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 [43]. Another vulnerability is the Stagefright affecting all at the time Android users. The vulnerability made it possible through a malicious MMS to gain full control over the Smartphone [2].

All applications with accessibility from the web have the problem of managing both trusted and untrusted data. The trusted data comes from the application itself an is mostly the database. Untrusted data is any data that users can alter. The consequences of trusting untrusted data can be catastrophic and to minimize the risk of accidentally or un-

knowingly introducing security flaws into applications has a variety of tools and methodologies arose. One of these is taint tracking which attempts to secure an application through information flow control. Taint tracking works by analyzing and finding paths user input can take through a system into the sensitive code areas. Cases, where this is possible, will be flagged to let the developers know where further implementation in the form of sanitation is needed [30, 46]. The question is how useful taint tracker is for web applications. Can a dynamic taint tracker, analyzing information flow at runtime, be used as a security solution for production services?

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

The following sections of the chapter aim to specify the why and how behind the thesis. It starts with a section of *Problem* description and explanation of the thesis *Aim*. These sections are then followed by a *Delimitations* and *Methodology* sections.

1.1 Problem

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

Unwanted information disclosures is a growing problem and work toward protecting user data is needed. Two of the most common vulnerabilities in this area, according to the Open Web Application Security Project, is Injection attacks and Cross-Site Scripting caused by unproperly sanitized user inputs [27]. Is it possible to protect web applications by implementing a dynamic taint tracker to run and analyze the code at runtime? Till what extent will this protect the application and what disadvantages will this entail? Is the solution capable of being bundled into production services?

1.2 Aim

This thesis aims to implement and evaluate a dynamic taint tracker to combat confidentiality and integrity vulnerabilities in Java-based web applications. The taint tracker aims to allow tracking of taint for all datatypes as well as constructing a robust design for the logic to mark sources, sinks, and sanitizers. Concretely, we will implement and benchmark a dynamic taint tracker against Injection and Cross-Site Scripting vulnerabilities.

1.3 Contribution

The contribution of the thesis is continued research in the field of solutions to combat information disclosure vulnerabilities. This is done through implementing and evaluating a dynamic taint tracking solution that runs independently between the Java Virtual Machine and the web application.

1.4 Delimitations

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

The delimitations of the application are that it will only consist of a dynamic taint tracker and not, in any form, a static version. Development of the tool is in and for Java applications with the help of the bytecode instrumentation library Javassist. The focus lies upon enabling taint tracking for Strings, and other data types are enabled if time allows for it.

The evaluation of the implemented dynamic taint tracker is only conducted to evaluate the prevention of integrity vulnerabilities even though the application itself can combat confidentiality vulnerabilities. This is due to time limitations.

1.5 Methodology

To answer the problem statement in section 1.1 is a combination of qualitative and quantitative research used. The literature study represents the qualitative research where information about web application security, Injection attacks, Cross-Site Scripting, and dynamic taint tracking gathered, presented and discussed. The quantitative research is the evaluation of the implemented dynamic taint tracker and benchmarks evaluating performance overhead and security gain.

1.6 Ethics & Sustainability

The ethical and sustainability aspects of the thesis have for the majority positive impact where all users of the application will achieve a gain in some form, except for users with malicious intent. The goal is to combat existing security vulnerabilities in today's web applications which would help both the end users and the companies who are providing the services. The end users gain would be to secure their information and reduce the risk of becoming victims of information disclosure. The companies would gain by providing a more secure product, which in turn would lower the risk of scandals that could hurt their brand.

Usage of the implemented dynamic taint tracker 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 ethical aspect which could be problematic is the fact that the taint tracker gains possible access to all data used by the system. This means that much work into the taint tracker needs to be done to ensure that no malicious use of the taint tracker results in harmful system executions.

1.7 Outline

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

Chapter 2

Background

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

This Chapter will present background knowledge needed to comprehend the thesis. The chapter starts with a general description about *Web Application* structure and is followed by a section discussing common *Security Vulnerabilities* to web applications. After those follows two sections describing *Taint Tracking*. The last section is about the programing language *Java*.

2.1 Web Application

To make applications available for a broad set of people and make them accessible from now days almost everywhere do businesses 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 illustrated in Figure 2.1. The first tier is the presentation tier which is the visual components rendered by the browser. The second is the logic tire which is the brain of the application. The last and third tier is the storage, where the second tier can store data as needed [8].

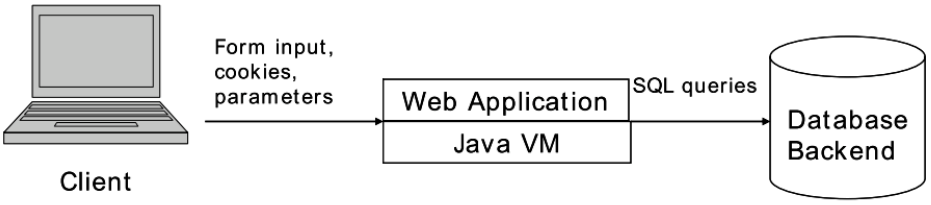

Figure 2.1: The three-tier web application architecture [15].

From Figure 2.1 can it be seen that tiers only communicate with the tire closest to themselves. This causes the second tier to become a safeguard for tier three where the valuable and possibly sensitive information is stored. This essential information is mostly needed for the application to provide its intended service. Such information might, for example, be name, email, personal number and credit card information [8].

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

2.1.1 Structured Query Language

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

2.2 CIA Triad

Discussions about application security rely on the CIA Triad which represents the three primary concepts in information security. These three are confidentiality, integrity, and availability and is seen in 2.2.

Confidentiality is rules that specify the access restrictions to the application. Integrity specifies that application data should be accurate and unaltered. Availability is about the ability to access the application and application data [7].

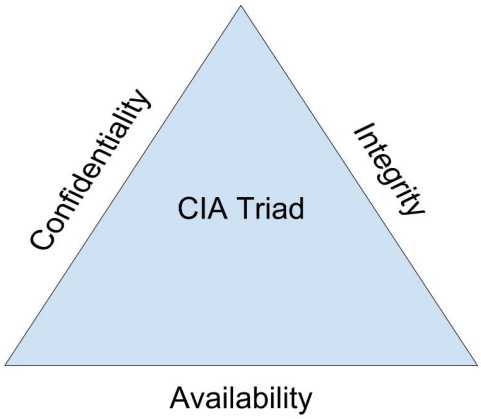

Figure 2.2: CIA Triad

2.3 Security Vulnerabilities

The organization Open Web Applications Security Project, known as OWASP, is an online community which aims to provide knowledge how to secure web applications [25]. OWASP has produced reports about the top 10 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 current 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 vulnerabilities are two vulnerabilities towards integrity and are Injection attacks and Cross-Site Scripting [27].

2.3.1 Injection Attack

The most common security risk is Injection Attacks [27]. 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 [39].

Injection Attacks can be divided into two different subgroups. These two subgroups are SQL Injection and Blind SQL Injection [39].

SQL Injection

SQL Injection is when a 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 a 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, 39].

Listing 2.1: Code Acceptable to SQL Injection

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

The query will 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). But what happens if the user input is *10 or 1 = 1*? This input would result in the query seen in Listing 2.2.

Listing 2.2: SQL Injection

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

This query results in an execution that always evaluates to true. The result of this will be that the query returns the whole table of users. This problem can be prevented in a couple of different ways. The first is through validation of input. By verifying user input as seen in Listing 2.3 can we protect the query from being vulnerable to SQL Injection.

Listing 2.3: Preventing SQL Injection through Verification

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

A second common alternative is to use SQL Parameters which handles 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: Preventing SQL Injection 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 that 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 took or what kind of error messages it returns. An example of the first is a 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, 39].

Listing 2.5: Time Based Blind SQL Injection

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

The second variant of 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, 39].

2.3.2 Cross-Site Scripting

Cross-Site Scripting has been a vulnerability since the beginning of the internet. One of the first Cross-Site Scripting attacks was created just after the release of JavaScript. 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 that is visible or typed into the web application. To prevent this form of attack where the standard of Same-Origin Policy introduced. Same-Origin Policy restricts JavaScript to only access content from its own origin [12, 34].

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 input of the message. Which would then later be reflected by the server itself, when displaying the message for other users, and thereby bypassing the Same-Origin Policy [12].

Dividing Cross-Site Scripting into three different subcategories is possible. 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 an unknowing user accesses. The malicious link will exploit a vulnerable input on the targeted web application and through the input reflect back malicious content to the user [39].

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 [39].

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 exploits the user [39].

2.4 Taint Tracking

Taint tracking, also known as taint analysis, taint checking and taint propagation, is a tool to analyze the flow of information in a domain [30]. 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 is possible to execute in two different forms: static and dynamic. The static is an evaluation tool which is done statically before runtime. Dynamic is a tool that is executed at runtime. There are pros and cons for both versions. The pros for static is the usability in the integrated development environment, and the pros for dynamic is a higher propagating taint accuracy.

Taint trackers operate by tracking data and actively blocking any that are trying to enter sinks without first been detainted through sanitation. Perl and Ruby are two programming languages which have adapted to use taint checking [31, 22]. There are some tools which enable taint checking for other platforms. TaintDroid [23] for the Android platform is one of them. This thesis will handle dynamic taint tracking and how it can increase the security of Java-based web applications.

Taint tracking contains four main tasks which are described in Table 2.1. The first is marking all data from sources as tainted. This is done through a taint flag attached to the input variables. This taint flag follows the input, which is the second task, throughout the application and propagates onto any other data it encounters. The third task, is the possibility to detaint data, but this is only done after the data have been sanitized through predefined sanitizers. Fourth and last task is checking the taint flags in areas called sinks which are entry points to sensitive code [30, 46]. The decision of what to do if a tainted variable tries to pass through a sink vary depending on the application, however, remedial actions should be conducted. These activities should

be, depending on application owners choice, logging events, throwing errors, or modifying the tainted values into safe predefined values.

Table 2.1: Core logic 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* a sink and *validate* a sanitizer. On row one, the input from the source is flagged tainted, and the taint propagates onto *userId*. The sanitizer on row two validates *userId* and removes the tainted flag. Lastly, the sink on row three 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. This would without a dynamic taint tracker 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: Taint Tracking

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

The above described dynamic taint tracking tool focuses on preventing malicious code from entering and causing harm to the application. These represent security policies restricting input from sources to pass through sinks without first being sanitized through validation. The same application could be used to enforce policies restricting sensitive

data to leave the system. It is done by marking access points to sensitive data as sources and unallowed exits of the system as sinks.

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 developer but still had a familiar C/C++ syntax. [26]. Today is Java one of the most common programming languages [13].

Java is a statically typed language which means that no variable can be used before being declared. These variables can be of two different types: primitives and reference to objects. Among the primitives does Java have support for eight. These are byte, short, int, long, float, double, boolean and char [33].

2.5.1 Java Virtual Machine

There exists a plethora of implementation of the Java Virtual Machine, but the official that Oracle develop is HotSpot [44]. 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 behind Java and the Java Virtual Machine. [9]. The idea behind the Java Virtual Machine was to have one language that executed the same on all platforms and then modify the Java Virtual Machine to be able 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.

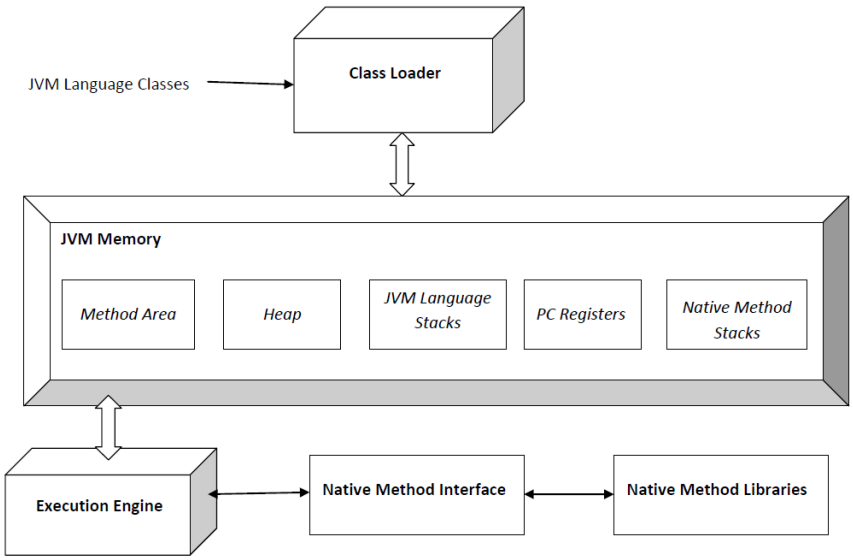

Figure 2.3: Java Virtual Machine Architecture

Figure 2.3 illustrates the architecture of the Java Virtual Machine. The ClassLoader loads the compiled Java code and adds it into the Java Virtual Machine Memory. The logic behind Java instrumentation lies in the ClassLoader. The ClassLoader will trigger the implemented Java Agent and allow for instrumentation of each Java file before being loaded into the Java Virtual Machine [47, 19].

2.5.2 Instrumentation

Java instrumentation is a way to modify the execution of an application without knowing nor the need of modifying the application code itself. Good use cases for Java instrumentation is, for example, monitoring agents, and event loggers. Instrumentation is an official Java package that provides services needed to modify the bytecode of program executions. It is conducted through implementing an Agent that will have the possibility to transform every class loaded by the ClassLoader before being used for the first time. Transformation of class files are on bytecode level, but there exist libraries that can help in this task. One of these libraries and the one used in this thesis is Javassist [18, 20].

There are some restrictions to instrument classes during runtime. Classes needed for the Java Virtual Machine need transformation before executing the Java application. This is because those classes load before the instrumentation agent. Those classes are the content of the base Java Runtime Environment in the rt.jar.

2.5.3 Javassist

There exist several libraries that can help the developer in the task of creating an instrumentation Agent. The help comes in libraries of high-level functions that later translates into bytecode that the Java Virtual Machine understands. The library used in this thesis is Javassist. Javassist stands for Java programming Assistant and is a bytecode engineering toolkit. Javassist provides two levels of API where the one used in this thesis provides the functionality of editing class files on source level which require no understanding of Java bytecode [20].

Chapter 3

Related Work

Todo: Bredare information. Ta med static och dynamic samt android.

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

This chapter presents the related work in the field. The first section is about *Dynamic Taint Trackers* which then follows by a section about **??**.

3.1 Dynamic Taint Trackers

Haldar, Chandra, and Franz [15] has written a report about dynamic taint tracking for Java 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 [15] 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 String by adding a taint flag and altered the methods to propagate the taint into the String class file.

Haldar, Chandra, and Franz [15] implementation cant be found but there exists two other dynamic taint trackers, Phosphor [32] and Dynamic Security Taint Propagation [11]. Both are open source projects

and developed for Java applications. Phosphor does however not support sanitizers, and Dynamic Security Taint Propagation cant build from its source code.

3.1.1 Phosphor

The construction of Phosphor [32] is done with the help of the Java bytecode manipulation library ASM [3]. Phosphor, based on the research conducted in the thesis, is the current state of art application in dynamic taint tracking. 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 [5].

3.1.2 Dynamic Security Taint Propagation

Dynamic Security Taint Propagation [11] is constructed with the help of the Java library AspectJ which enables aspect-oriented programming in Java [40]. 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.

Chapter 4

Implementation

Todo: Skapa en bild som illustrerar achitecturen på applicationen. Förklara propageringslokien ordentligt.

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

This Chapter presents the fundamental parts of the implementation process of the dynamic taint tracker. The chapter starts with a section describing *Policies* enforced by the application. This section is then followed by *Software Implementation*.

4.1 Policies

The development of the dynamic taint tracker relies on the tasks described in Table 2.1. These are tainting, detainting, propagating taint, and assert non-tainted. However, to implement the logic need 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 *confidentiality* and *integrity*.

4.1.1 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 information.

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

4.1.2 Integrity

The integrity policy defines that users may not modify data which they do not have permission to alter. This goal is to ensure prevention of malicious usage of the application where attackers aim to destroy application data. This gives us the policy:

- No information shall be altered without the users having permission to do so for the information.

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

4.1.3 Taint Checking

The policies above are enforced through validation of user input from sources and any data it has come in contact with. By enforcing these should preventions of confidentiality and integrity volubilities be reduced severely.

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

To enable tracking of taint is an implementation of taint propagation needed. The goal is to support all data type. This is, however, a complex problem. Instrumentation of classes is decently, but instrumentation of primitives and arrays are a rather complex problem. However, the principal behind the tracking is the same for all data types.

Below are rules defining when taint variables should propagate:

- Data resulting in a copy, subset or combination of tainted data shall be tainted.
- Data disclosing information about tainted data shall be 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" ?* [48] and *Searching for Code in J2EE/Java* [35]. These web pages present sources, sinks, and sanitizers from their experience with developing web applications.

4.3 Software Implementation

The implementation of the dynamic taint tracker is divided into three subprojects. These three are Agent, Xboot, and Utils.

Agent Project that transforms classes loaded at runtime into sources, sinks or sanitizers.

Xboot Project that loops through all classes in rt.jar and transforms into sources, sinks or sanitizers.

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. Therefore, to remove duplications of code is al logic of transforming classes extracted from Xboot and Agent and placed into the Utils project.

The implemented dynamic taint tracker supports tracking of taint for the classes: String, StringBuilder and StringBuffer. The goal was to implement tracking for all classes. However, this took more time than expected.

4.3.1 The Utils Project

The Utils project includes the core logic of marking methods and classes as sources, sinks, and sanitizers. It works by taking a class as an argument that is to be checked if it qualifies for any of the three below criteria.

- Is same class as the defined source, sink or sanitizer.
- Implements interface of the defined source, sink or sanitizer.

- Extends defined source, sink or sanitizer class (recursive call. Checks all in the list for each extended class).

If a class fulfills any of the three criteria will the list of defined method correlating to either source, sinks or sanitizer be used, and instrumentation of the methods will be conducted.

The instrumentation of the method works differently depending on if it is a source, sink or sanitizer. Where 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 parameters are tainted. If anyone of them is, then a taint exception occurs, and the remedial action of changing the value to a predefined value, where the value is an empty string, and logging the event used.

should conduct. 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 file to the front of the bootstrap classpath. Making the Java Virtual Machine use our modified versions of the base Java Runtime Environment [17] before the original version.

Another problem is that instrumentation of primitives and arrays not possible. This causes a problem since it opens the ability to miss tracking of tainted data if they ever pass through a byte- or char array. The

solution that can solve this is to create shadow as Bell and Kaiser [5] did while creating Phosphor [32].

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 and Javassist, bytecode level API, is suitable for this. [20]

Chapter 5

Evaluation

Todo: Ta med vilken typ av policies som ska uppfyllas per application!!! Förklara att det är tidskrävande och därav körs bara 4 applicationer. Lägg till bild på hur ZAP kommer in i architecturen.

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

This section describes the conduction of the benchmarking of the implemented dynamic taint tracker. The chapter starts with a description of the *Test Environment* followed by a detailed description about the *Benchmarking*

5.1 Test Environment

The execution of the benchmarking is conducted on an Asus Zenbook UZ32LN. No other programs were running while benchmarking was 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

Each execution of benchmarks executes two times. One without and one with dynamic taint tracking. The first is to acquire the baseline of the application. The second is to acquire how dynamic taint tracker affects the execution of the application.

5.2.1 Applications

To detect security vulnerabilities in the applications has OWASP Zed Attack Proxy [29] known as ZAP ben used. ZAP is an open-source security scanner for web applications which is widely used in the penetration testing industry.

To only scan applications for vulnerabilities of interest is a new policy specified in the ZAP application. 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 dynamic taint tracker (DTT) 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 starts with spidering the application to detects all possible entries to the system. If the application requires authentication to ac-

cess parts of the web application is this information added to the ZAP context and then is the spider executed again to find all possible new entries. After these steps are the scanning of the application activated and the security vulnerabilities are stored in a report file.

The benchmarking was conducted on four web applications. Each application is Java-based and is deliberately implemented with security vulnerabilities such as Injection Attack and 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 was created as part of the Griffin Security Project [14] at Stanford University and contains 96 test cases and 46407 lines of code. This thesis uses version 1.08 of the application [37, 38].

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 [28].

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 [49] 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 [36].

Ticketbook

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

plication consist of 13849 lines of code and version 0.9.1-SNAPSHOT is used [45, 24]

5.2.2 Performance Overhead

To evaluate the time and memory overhead is The DaCapo Benchmark Suit [42] used. DaCapo is a set of applications constructed specifically for Java benchmarking. This thesis uses the 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 *The benchmarks* [41].

Table 5.2: Descriptions for each application in The DaCapo Benchmark Suit taken from *The benchmarks* [41]

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 which executes each application ten times both with and without dynamic taint tracking. To isolate each iteration is a unique process spawned per test case execution. This process will then run the application in a child process which will be evaluated for time and memory. This information is then passed back to the main thread where all data is aggregated.

Chapter 6

Result

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

This chapter presents the results of the conducted evaluation. Appendix A contains raw data and metrics over data that may not be shown in this chapter. The chapter starts with presenting the results from the *Performance Overhead* evaluations where the parameters time and memory is measured. Next and the last section is *Applications* where Java applications have been evaluated measuring security vulnerabilities with and without Dynamic Taint Propagation.

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 [37]. 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 [37] application results in a 100% prevention rate.

Table 6.1: Security Vulnerabilities Detected by dynamic taint tracker (DTT) in Stanford SecuriBench Micro

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

Table 6.2 shows the vulnerabilities from running InsecureWebApp [28] 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 [28] 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 dynamic taint tracker (DTT) in InsecureWebApp

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

The results from evaluating the application SnipSnap [36] is seen in Table 6.3. 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 6.3: Security Vulnerabilities Detected by dynamic taint tracker (DTT) in SnipSnap

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

Table 6.4 shows the vulnerabilities from evaluating Ticketbook [45]. 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 6.4: Security Vulnerabilities Detected by dynamic taint tracker (DTT) in Ticketbook

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

6.2 Performance Overhead

The results from benchmarking the application on DaCapo Benchmark Suit [42] 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 ap-

plication, however, was Batik with an overhead of 432.2%. The average overall is 162.9%.

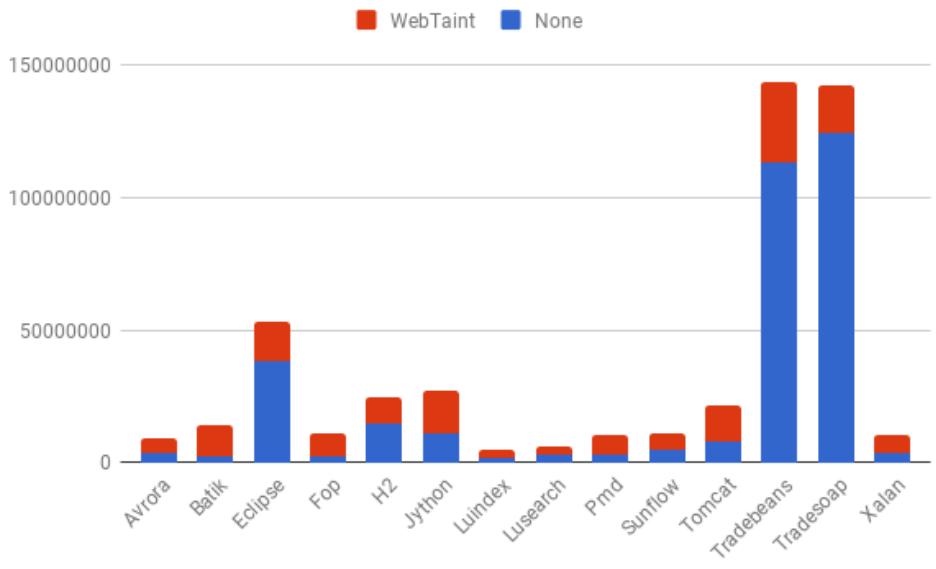

Figure 6.1: Average Added Time in Microseconds

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%.

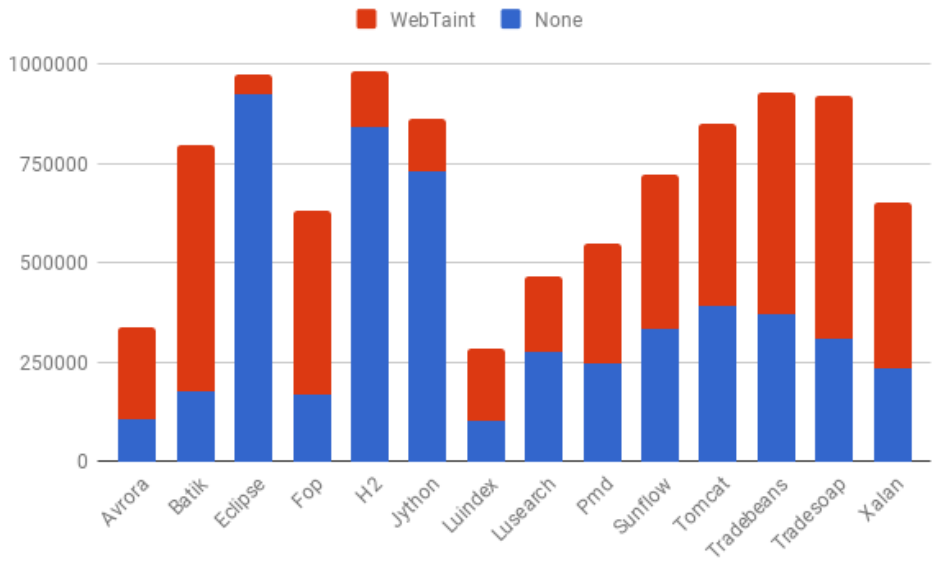

Figure 6.2: Average Added Memory in Kilobytes

Chapter 7

Discussion

Todo: Uppdatera till senaste (SISTA SOM GÖRS)

This chapter contains the discussions about the implemented dynamic taint tracker and how well it performs. The chapter starts with a general presentation about the implementation. This follows by a section comparing it with **??**. This is then followed by *Sources, Sinks & Sanitizers* discussions. Lastly is there two sections about *Taint Tracking* 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. I 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. I can see it going two 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. I 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 its place as a solution to secure web applications. One of these is to conduct comprehensive work about 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 a more detailed taint processing where sanitizers only capable of sanitizing a specific type of data do not mark other types of data as detainted.

Work towards optimization of the applications is also in need. Bot towards minimizing the added time overhead as well as the added memory overhead. To enhance the coverage of the tool should expansions of data types supporting tracking of taint carry out as well. The two most important data types are char and byte arrays.

Another way to enhance the coverage of WebTaint is to develop the application further to support implicit taint flow. WebTaint does at the moment only support explicit taint flows where taint propagates if the resulting variables are directly dependent on the tainted variables. For example would x in $x = y + 1$ become tainted if y is tainted. Implicit taint flow would enable taint to propagate implicitly where an example is that x would be tainted if $if(y)x = 1$ and y is tainted.

There is also the need for further research and evaluation for the field and the implemented taint tracker. Trial runs where WebTaint runs for a more extended period would be of interest. The interest lies in getting

an insight on how the percentage time and memory overhead changes compared to the results of this report.

Chapter 9

Conclusion

We implemented and evaluated a dynamic taint tracker for Java-based web applications called WebTaint. The application's goal is to combat integrity and confidentiality vulnerabilities. We can see an improvement in security when applying WebTaint to applications. However, there are drawbacks regarding added overhead in the form of time and memory. These are at the moment quite significant but could lower through optimizations. This causes the dynamic taint tracker not to be optimal if used in time- or memory sensitive domains.

Bibliography

[1] B. Allen et al. "Software As a Service for Data Scientists". English. In: *Communications of the ACM* 55.2 (2012). ISSN: 00010782.

[2] "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.

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

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

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

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

[16] *IaaS, PaaS, and SaaS: The Good, the Bad and the Ugly)*. URL: https://www.stratoscale.com/resources/article/iaas-paas-saas-the-good-bad-ugly/ (visited on 05/28/2018).

[17] *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).

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

[19] *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).

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

[21] Newcombe Lee. "Security and Software as a Service". eng. In: *Securing Cloud Services - A Pragmatic Approach to Security Architecture in the Cloud*. IT Governance Publishing, 2012, pp. 1–3. ISBN: 9781849283977. URL: https://app.knovel.com/hotlink/pdf/rcid:kpSCSAPAS4/id:kt00BIWD9H/securing-cloud-services/security-software-service?kpromoter=Summon.

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

[23] 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.

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

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

[26] *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).

[27] 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.

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

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

[30] 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.

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

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

[33] *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).

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

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

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

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

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

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

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

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

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

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

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

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

[46] 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.

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

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

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

[50] *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

In this appendix are tables containing row data not included in the thesis presented. These tables are Table A.1 and A.2 which contains average, min and max values from executing overhead performance benchmarks.

Table A.1: Time Overhead (ms)

	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 Overhead (kilobytes)

	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