Dynamic Taint Tracking for Domain-Driven Security (DRAFT)

FREDRIK ADOLFSSON

Master in Computer Science
Date: May 23, 2018
Supervisor: Musard Balliu
Examiner: Mads Dam
Swedish title: Dynamisk taint spårning för domändriven säkerhet
School of Computer Science and Communication

Todo: Fix "FIX ME!!!" reference for unpublished book Secure by Design.

Abstract

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 Tracking tool 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 dose.

Sammanfattning

Todo:

Contents

Chapter 1

Introduction

The creation of the World Wide Web (web) has caused a significant impact on today's society [54]. 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. The information sharing is a weakness in the same manner as it is a strength. The web application is 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 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. The Open Web Applications Security Project, known as OWASP, is an online community which aims to provide knowledge about how to secure web applications [28]. OWASP has produced reports about the top 10 security risks for a web application, and the latest published in 2017. In this report was Injection Attacks number one and Cross-Site Scripting number seven [30, 28, 9].

To minimize the risk of accidentally introducing security flaws into the application has a variety of tools and methodologies created. One of these is Dynamic Taint Tracking which marks input from the user as tainted through a taint variable attached to the variable representing the input. This taint variable follows the input throughout the application and propagates onto the other variables it encounters. It is possible to detaint the input, and this is after the input is validated. The assertion of non-tainted values occurs in sinks where tainted variables are prevented from executing [33, 49].

One of the methodologies coined is the programming paradigm Domain-Driven Security. Domain-Driven Security aims to secure applications by focusing on the core domain models and making sure that validation of the value primitives is correct [53, 22].

The following sections of the chapter aim to specify the why and how behind the conduction of the thesis. It starts with a section of *Definitions* followed by *Problem* description and explanation of the thesis *Aim*. These sections is then followed by a *Delimitations* section. Lastly, is there a section about the *Methodology* behind the thesis.

1.1 Definitions

Definition 1.1.1. Application is a computer process constructed to solve one or more tasks for users.

Definition 1.1.2. Web Application is an application deployed with accessibility from the web.

Definition 1.1.3. Taint marking data with a flag indicating the possibility to be harmful to the application.

Definition 1.1.4. Detaint denotes the process of removing the taint flag from a value and therefore marking the value as safe to the application.

Definition 1.1.5. Source denotes an entry point to the system where the input is possibly malicious.

Definition 1.1.6. Sink denotes entry point to sensitive code areas.

Definition 1.1.7. Sanitizer denotes method that validates and sanitizes data to be safe to the system.

Definition 1.1.8. Domain is explained in Secure by Design [26] as part of the real world where something happens.

Definition 1.1.9. Domain Model is a fraction of the domain where each model has a specific meaning.

1.2 Problem

> *How can the implementation of a Dynamic Taint Tracking tool enforce the security gains of Domain-Driven Security?*

Unwanted information disclosure is a growing problem. Work towards protecting user data is needed, and Domain-Driven Security has been proven to secure applications from Injection and Cross-Site Scripting attacks. Is it possible to achieve the security gains of Domain-Driven Security through applying Dynamic Taint Tracking to web applications? What would the potential drawbacks and advantages be?

1.3 Aim

This thesis will implement and evaluate a Dynamic Taint Tracking tool to prevent confidentiality and integrity vulnerabilities in Java-based web applications. The thesis will also evaluate the security benefits of Domain-Driven Security, a programming paradigm which has been proposed to combat confidentiality and integrity vulnerabilities. Concretely, we will benchmark our Dynamic Taint Tracking tool against injection, cross-site scripting, and information disclosure vulnerabilities.

1.4 Delimitations

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

Delimitations for the application is 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 application with the help of the bytecode instrumentation library Javassist.

1.5 Methodology

The methodology of this thesis is a combination of qualitative and quantitative research. The qualitative research represents the literature study where information about web application security, Dynamic Taint Tracking and Domain-Driven Security is gathered, presented and discussed. The quantitative research is the evaluation of the implemented Dynamic Taint Tracking tool. The benchmarks will evaluate performance overhead and security gain from preventing possible attacks.

Chapter 2

Background

This Chapter will present background knowledge needed to comprehend the conduction of the thesis. The chapter starts with a general description about *Web Application* structure and is followed by a section discussion common *Security Vulnerabilities* to web applications. After those follows two sections describing *Dynamic Taint Tracking*, *Domain-Driven Design*, and *Domain-Driven Security*. The last section is about *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. 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 [7]. An illustration of the three-tier architecture can be seen in figure 2.1.

It can be seen in figure 2.1 that the 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. The storage tier contains all the essential information the appli-

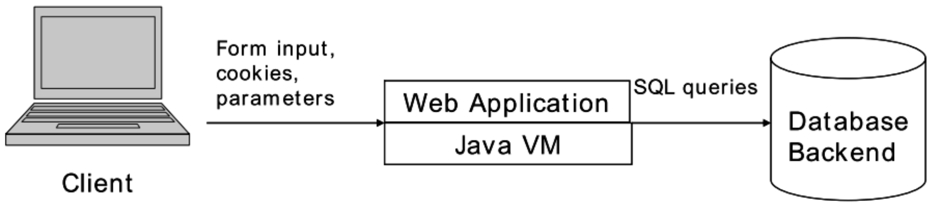

Figure 2.1: The three-tier web application architecture [17].

cation needs to provide its intended service. Such information might, for example, by name, email, personal number and credit card information [7].

The scope of the thesis lies in tier two. 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 Security Vulnerabilities

The organization Open Web Applications Security Project, mostly known for its shortening OWASP, is an online community which aims to provide knowledge how to secure web applications [28]. OWASP has produced reports about the top 10 security risks for a web application, and the latest was published in 2017. The report contains information about the ten most common application security risks that for the current year. Information such as how the security risk is exploited and possible prevention method is also presented. This thesis will look at security risk number one and eight which is Injection Attacks and Cross-site Scripting [30].

2.2.1 Injection Attacks

The most common security risk is Injection Attacks [30]. 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 [43].

Injection Attacks can be divided into two different subgroups. These two subgroups are SQL Injection and Blind SQL Injection [43].

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 [7, 43].

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, notated with *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 user 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 the input. By verifying the user input as seen in listing 2.3 can we protect the query from being accessible 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 [7, 43].

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, which is by analyzing the error messages, and depending on what they return to build an image of the wanted answer. This is mostly done by testing the different combination of true and false questions [7, 43].

2.2.2 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 information from. 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 [14, 37].

The introduction of the Same-Origin Policy did not stop the attackers from performing Cross-Site Scripting attacks. The next wave of attacks was mostly towards chat rooms where it was possible to inject malicious 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 [14].

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 clicks. The malicious link will exploit a vulnerable input on the targeted web application and through the input reflect back content to the user [43].

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 [43].

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 exploit the user [43].

2.2.3 CIA Triad

Discussions about application security often rely on the CIA Triad which represents the three primary concepts in information security. These three are confidentiality, integrity, and availability. Confidentiality is rules that specify the access restrictions to the application. Integrity specifies that application data should be accurate and not altered. Availability is about the ability to access the application and application data [6]. This thesis focuses on confidentiality and integrity vulnerabilities and how we can prevent them.

Injection Attacks and Cross-site Scripting could be attacks both towards the confidentiality and integrity of systems. They are attacks towards confidentiality when the attacker intends to gain restricted information such as user data. Integrity attacks are conducted when for example Injection Attacks are used to redirect users to malicious websites.

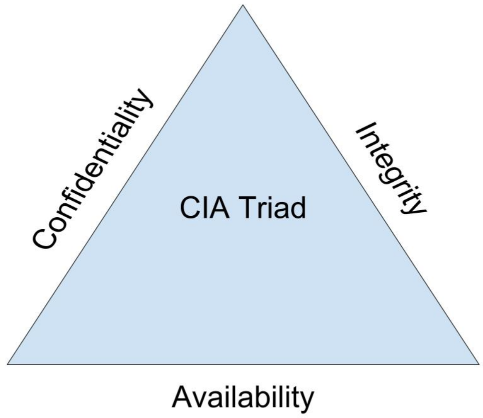

Figure 2.2: CIA Triad

2.3 Dynamic 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 [33]. The goal of taint tracking is to prevent possible attacks such as Injection Attacks and Cross-Site Scripting by enforcing the usage of sanitizers on input data. Taint tracking is possible to execute in two different ways: static and dynamic. The static is an evaluation tool which is done statically before runtime. Dynamic is a tool that is executed at runtime. The tool works by tracking data and actively blocking any data that is trying to enter sinks without being detainted through sanitation first. Perl and Ruby are two programming languages which have adapted to user taint checking [34, 23]. There are some tools which enable taint checking for other platforms such as TaintDroid [24] for the Android platform. This thesis will handle Dynamic Taint Tracking and how it can increase the security of a web application.

Taint tracking works by marking untrusted input at sources as tainted. This is done through a taint flag attached to the input. This taint flag follows the input throughout the application and propagates onto any other data it encounters. It is possible to detaint tainted data, but this is only done after the data have been sanitized through validation. The taint flags are checked in areas called sinks which are markings for entry points to sensitive code [33, 49]. The decision of what to do if a tainted variable tries to pass through a sink vary depending on the application. However, the typical reaction is to either log information about the execution or throw an execution.

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 taint flag. Lastly, the sink on row three executes 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 Tracking tool result in giving the malicious user the entire list of Users. With a Dynamic Taint Tracking tool, 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 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.

2.4 Domain-Driven Design

Domain-Driven Design is a thought process and methodology to follow in every step of the process [13]. In *Domain-driven design reference: definitions and patterns summaries* do Evans [12] describe Domain-Driven Design through three core ideas:

- Focus on the core domain.
- Explore models in a creative collaboration of domain practitioners and software practitioners.
- Speak a ubiquitous language within an explicitly bounded context.

The core domain is the part of the product that is most important and often is the main selling point compared to other similar products [27]. A discussion and even possible a document describing the core domain is something that will help the development of the product. The idea is to keep everybody on the same track and head in the same direction [13].

The second idea is to explore and develop every model in collaboration between all domain practitioners, who are experts in the given domain, and software developers. This ensures that essential knowledge needed to create the product successfully is communicated back and forth between the two parties [27]. The third idea is necessary to enable and streamline the second. By using a ubiquitous language, will miscommunication between domain and software practitioners be minimized and the collaboration between the two parties can instead focus on the essential parts which are to develop the product [12].

Evans [12] do as well argue about the weight of clearly defining the bounded contexts for each defined model, and this needs to be done in the ubiquitous language created for the specific product. The need of this exists because of the otherwise high risk of misunderstandings and erroneous assumptions in the collaborations between the different models [27].

2.5 Domain-Driven Security

Wilander [53] and Johnsson [22] created 2009 a blog post each in a synchronous manner where they together introduce the concept of Domain-Driven Security to the public. They describe Domain-Driven Security as the intersection between Domain-Driven Design and application security. Domain-Driven Security focuses on the importance of input validation and developing and maintaining domain models which reflect the product correctly. To enforce validations is value primitives introduced. Doing so minimizes the risk of accidentally propagating erroneous data. Value primitives are a minor modification of Domain-Driven Designs domain value. [53, 22, 1, 42].

2.6 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. [29]. Today is Java one of the most common programming languages [15].

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. These are primitives and references to objects. Among the primitives does Java have support for eight. These are byte, short, int, long, float, double, boolean and char [36].

2.6.1 Java Virtual Machine

There exists a plethora of implementation of the Java Virtual Machine, but the official that Oracle develop is HotSpot [47]. One of the core ideas with Java during its development was "Write once, run anywhere." The slogan was created by Sun Microsystems which at the time were the company behind Java and the Java Virtual Machine. [8]. 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 the components heap storage, stack, program counter, method area, and runtime constant pool.

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 [50, 20].

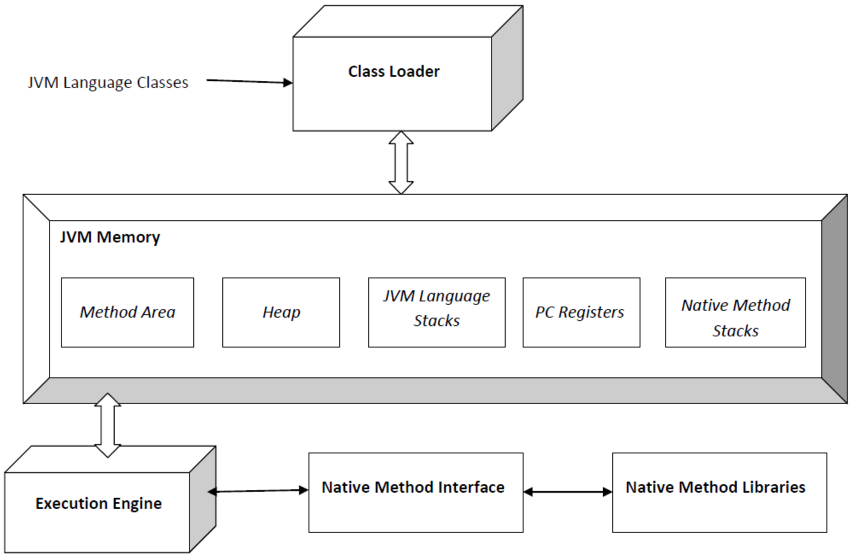

Figure 2.3: Java Virtual Machine Architecture

2.6.2 Instrumentation

Java instrumentation is a way to modify the execution of an application on the Java Virtual Machine 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 official Java package that provides services for modifying the bytecode of the program execution. Instrumentation works by implementing an Agent that will have the possibility to transform any class loaded by the application before being used for the first time. Transformations 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 [19, 21].

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

2.6.3 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 will understand. 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 [21].

Chapter 3

Related Work

Stendahl [42] and Arnör [1] have both, in their thesis's, concluded that Domain-Driven Security help to prevent security vulnerabilities into an application. Stendahl [42] thesis, written in 2016, evaluated if Domain-Driven Security can prevent Injection Attacks and Cross-Site Scripting. He reasoned that there is a security gain towards Injection Attacks and Cross-Site Scripting by following the Domain-Driven Security methodology. The gained security comes from proper validation of variables before propagating the data into the value primitives. Arnör [1] followed similar reasoning where he discussed the mitigation of DDoS attacks by using Domain-Driven Security.

Haldar, Chandra, and Franz [17] has written a report about Dynamic Taint Tracking in Java where they try 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 [17] ran their benchmarks on OWASP's project WebGoat [5] but acknowledged in their report that benchmarks of real-world web applications need conducting.

It exists two Dynamic Taint Tracking tools, Phosphor [35] 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.

The construction of Phosphor [35] is done in the Java bytecode manipulation library ASM [2]. Phosphor, based on the research conducted in the thesis, is the current state of art application in Dynamic Taint Tracking. The application solved propagation of primitive and array taint 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 after each primitive and array. Each method in the application is also instrumented to pass shadow variables together with the un-instrumentable object. This solution makes Phosphor capable of propagating taint for all Java data types [4].

Dynamic Security Taint Propagation [11] is constructed with the help of the Java library AspectJ which enables aspect-oriented programming in Java [44]. Dynamic Security Taint Propagation only propagates taint for the String, StringBuffer, and StringBuilder classes. The propagation works by creating aspect-oriented events that trigger the propagation of taint, tainting sources and assertions that tainted values do not pass through sinks.

Chapter 4

Implementation

This Chapter presents the fundamental parts in the process of implementing the Dynamic Taint Tracking tool. The chapter starts with a section describing the *Policies* of the tool. This section is then followed by *Software Architecture* and *Notable Problems*.

4.1 Policies

The development of the Dynamic Taint Tracking tool relies on tainting, detainting, propagation logic and asserting negative taint. However, to implement the logic of the application need the security policies first be defined. Security policies are principles or actions that the application strives to fulfill [3]. In the application developed in this thesis will these be based on two different aspects. These are *confidentiality* and *integrity*.

4.1.1 Confidentiality

The confidentiality policies entailed that data given to the user should only be data that the user have the right to access. This gives us the policy below.

- No information shall be released to users without permission.

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

4.1.2 Integrity

Integrity entails that users may not modify data which they do not have permission to alter. This gives us the policy below.

- No information shall be altered without permission.

This entails that no information from sources shall gain access to a sink without first being sanitized.

4.1.3 Taint Checking

The policies above will be enforced by forcing validation of data that are or have been in contact with data coming from a source before they enter a sink. By enforcing this rule should preventions of confidentiality and integrity volubilities be reduced severely.

The policies above can also be combined with tainting policies. These are presented below.

- Data passing through sources going into the domain should be marked tainted.
- No tainted data is allowed to pass through a sink.
- Data can only be detainted through validation.

4.1.4 Taint Propagation

To enable the tracking of taint in the system is a complete implementation of taint propagation needed. The ultimate goal would be to have support for propagation of taint for each class and data type. This is, however, a complex problem. Instrumentation of classes is decently, but instrumentation of primitives and arrays is a rather complex problem. However, the principal behind the propagation is the same for all data types.

Below are rules defining when taint variables should propagate.

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

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" ?* [51] and *Searching for Code in J2EE/Java* [38]. These web pages present sources, sinks, and sanitizers from their experience with developing web applications.

4.3 Software Architecture

The implementation of the Dynamic Taint Tracking tool 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 Tracking tool supports propagation of taint for the classes: String, StringBuilder and StringBuffer. The goal was to implement propagation 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 remedial action 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.

4.4 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 [18] 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 propagation 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 [4] did while creating Phosphor [35].

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. [21]

Chapter 5

Evaluation

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 Performance Overhead

To evaluate the time and memory overhead is The DaCapo Benchmark Suit [46] 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 5.1 contains a description for each application. Summary is taken from *The benchmarks* [45].

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.

5.2.2 Applications

To detect security vulnerabilities in the applications has OWASP Zed Attack Proxy [32] 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 5.2 are used.

Every scan starts with spidering the application to detects all possible entries to the system. If the application requires authentication to access parts of the web application is this information added to the ZAP

Table 5.1: Descriptions for each application in The DaCapo Benchmark Suit taken from *The benchmarks* [45]

Avrora Simulates a number of programs run on a grid of AVR micro-controllers.

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.

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 SQL Injections and Cross-Site Scripting. These four Java web applications are presented in the sections below.

Table 5.2: 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

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 [16] at Stanford University and contains 96 test cases and 46407 lines of code. This thesis uses version 1.08 of the application [40, 41].

InsecureWebApp

InsecureWebApp is a deliberately insecure web application developed by OWASP to show possible security vulnerabilities and what harm they can cause to a web application. The project consists of 2913 lines of code and version 1.0 is used [31].

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 [52] where users can sign up and contribute by writing posts. The application consists of 566173 lines of code and version 1.0-BETA-1 is used in this thesis [39].

Ticketbook

Ticketbook is deliberately insecure web application developed by Contrast Security to show the power of one of their security tools. The application consist of 13849 lines of code and version 0.9.1-SNAPSHOT is used [48, 25]

Chapter 6

Result

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 Performance Overhead

The results from benchmarking the application on DaCapo Benchmark Suit [46] 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.

6.1.1 Time

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

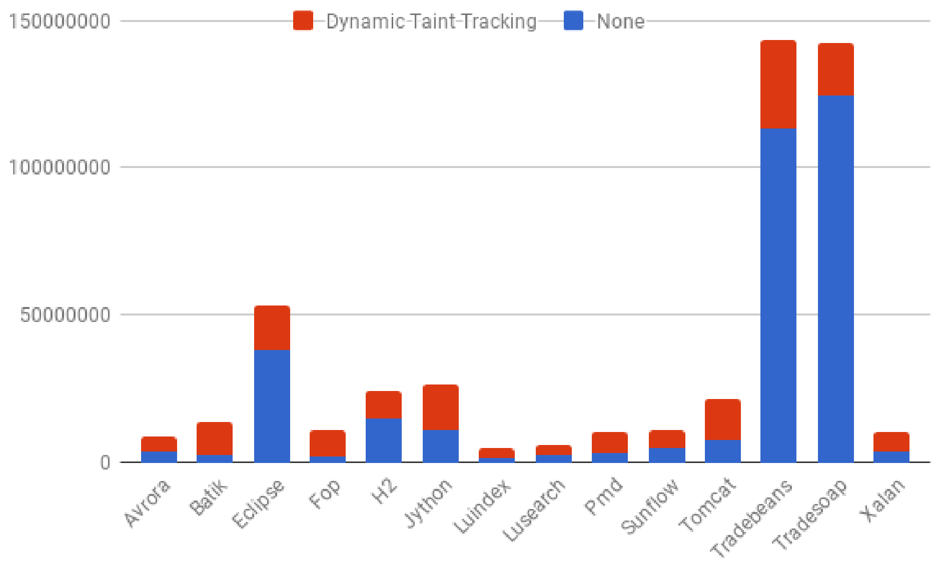

Figure 6.1: Average Added Time in Microseconds

6.1.2 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%.

6.2 Applications

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

Table 6.1 shows the vulnerabilities from evaluating Stanford SecuriBench Micro [40]. In the table can we see that the most common vulnerability

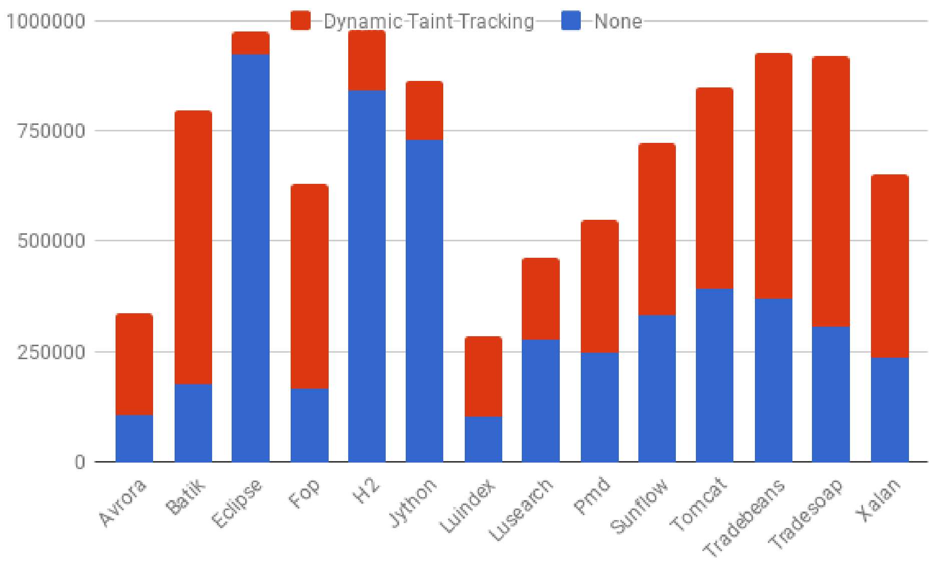

Figure 6.2: Average Added Memory in Kilobytes

is reflected Cross-Site Scripting where 71 vulnerabilities are present. Second most common is SQL Injection with 20 and the least common with one vulnerability is Buffer Overflow. By enabling Dynamic Taint Tracking on the Stanford SecuriBench Micro [40] application results in a 100% prevention rate.

Table 6.1: Security Vulnerabilities Detected by 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 [31] 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 [31] 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 [39] 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 [48]. The most common vulnerability was Cross-Site Scripting with 14 occurrences. SQL Injection was the least with one. The prevention rate of SQL Injection was 100% and for Cross-Site Scripting 71.4%. The overall prevention rate is 73.3%.

Table 6.4: 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

Chapter 7

Discussion

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 *Domain-Driven Security*. This is then followed by *Sources, Sinks, and Sanitizers* discussions. Lastly is there two sections about *Taint Propagation* and *Methodology of Evaluation*.

From the results can we see that using the Dynamic Taint Tracker adds a performance overhead. This overhead comes from instrumenting the classes and from the added operations for propagating taint. The most significant impact on this, for time overhead, comes from the instrumentation which is seen in both Figure 6.1. The instrumentation is only done once which makes applications with longer runtime less impacted by the added time overhead. This 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 here have about the same memory overhead as the average which is 142.7%. For the memory overhead to be the same ratio for all applications is understandable since the added memory overhead comes from the jar file containing the instrumented rt.jar that is appended to the bootstrap classpath and from adding size to every String, StringBuilder and StringBuffer.

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 destroy the user experience of a web application if the added time for events to happen takes a long time.

7.1 Domain-Driven Security

As said prior, Domain-Driven security has been proven to mitigate Injection and Cross-Site Scripting attacks. By looking at the result of this thesis, can we also see that Dynamic Taint Tracking mitigates the same security vulnerabilities. Dynamic Taint Tracking is, however, compared to Domain-Driven, only an application that enforces validation of variables that have been in contact with untrusted sources. How they are validated does not matter as long as the method used is defined as a sanitizer by the Dynamic Taint Tracker. Domain-Driven Security, however, utilizes value primitives to secure validation and sanitation. They are a single point of validation which lowers the complexity of redefining or expanding the logic of the application. Dynamic Taint Propagation could be altered to enforce validations inside the value primitives constructors. It could work by altering all sanitizers to the only detaint when called from inside a value primitives constructor. It would force the user to use value primitives.

7.2 Sources, Sinks, and Sanitizers

One part of the thesis that felt like a minor part before the actual conduction was defining sources, sinks, and sanitizers. This work could be a thesis in itself. The solution to this was to aggregate what could find others defining as sources, sinks, and sanitizers. It is however not an optimal solution. For a taint tracker to become widely used, do I believe that every service provider for Java applications and libraries should define the sources, sinks, and sanitizers in their implementation. The users could then subscribe on lists defining sources, sinks, and sanitizers depending on the application the tracker will analyze.

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 execution to execute. The remedial action could even be to sanitize the data with predefined sanitation functions.

Another thing of interest would be to introduce multiple taint types. It could be used to ensure that data from sources sanitizes with the correct sanitizer depending on the source type. Since data from one type of source might not be harmful to all types of sinks and all types of sanitizers might not be possible to sanitize the data for all sinks correctly.

7.3 Taint Propagation

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

However, the results prove that for web applications do these classes cause a significant difference and taint tracking for them is still better than for none. Nevertheless, the optimal solution would be with complete integration of all Java data types. Just like Phosphor, but with the ability to sanitize variables.

7.4 Methodology of Evaluation

The objective of the thesis was from the beginning to implement the Dynamic Taint Tracker and benchmark it in comparison to Dynamic Security Taint Propagation and Phosphor. This was sadly not possible since the first 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 a lot that needs doing before Dynamic Taint Tracking can take its place as an action to secure web applications. The most important is to conduct a comprehensive work about sources, sinks, and sanitizers. Implementation of specifying different taint labels, where sanitizers might only detaint some of them, would be of interest as well.

Work towards optimization of the applications needs conduction as well. Bot towards minimizing the added time and memory overhead.

To enhance the coverage of the tool should expansions of data types supporting propagation of taint carry out. The two most important data types are char and byte arrays.

Even though we see benefits from the conducted experiments is there always need for further benchmarking. Trial runs where the Dynamic Taint Tracking runs for a longer time would be of interest. The interest lies in getting an insight on how it could be used to patch the taint exceptions based on logging information about taint exceptions caught. How would this affect the development of the web application? Would developer stop focusing on sanitation and verification because the Dynamic Taint Tracking will tell them what to fix?

Chapter 9

Conclusion

We can see an improvement in security when applying the Dynamic Taint Tracker to a Java-based web application. However, there are drawbacks regarding added overhead in the form of time and memory. These are at the moment markedly noticeable but could lower through optimization. This causes the Dynamic Taint Tracker not to be optimal if used in time- or memory sensitive domain, at least not before optimizations.

Bibliography

[1] Johan Arnör. "Domain-Driven Security's take on Denial-of-Service (DoS) Attacks". In: (2016), p. 54. URL: http://kth.diva-portal.org/smash/get/diva2:945831/FULLTEXT01.pdf.

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

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

[4] 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.

[5] *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).

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

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

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

[9] Michael Cross. *Developer's guide to web application security*. eng. Rockland, MA: Syngress Publishing, 2007. ISBN: 1-281-06021-6.

[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] Eric Evans. *Domain-driven design reference: definitions and patterns summaries*. Dog Ear Publishing, 2015.

[13] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. eng. Boston, Mass.: Addison-Wesley, 2004. ISBN: 0-321-12521-5.

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

[18] *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).

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

[20] *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).

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

[22] Dan Bergh Johnsson. *Dear Junior - Letters to a Junior Programmer: Introducing Domain Driven Security*. 2009. URL: `http://dearjunior.blogspot.se/2009/09/introducing-domain-driven-security.html` (visited on 01/25/2018).

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

[24] 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.

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

[26] FIX ME!!! *Secure by Design*. FIX ME!!!, 2018.

[27] Scott Millett. *Patterns, principles, and practices of domain-driven design*. Wrox, a Wiley brand, 2015.

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

[29] *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).

[30] 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.

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

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

[33] 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.

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

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

[36] *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).

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

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

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

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

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

[42] Jonas Stendahl. "Domain-Driven Security". In: (2016), p. 39. URL: http://kth.diva-portal.org/smash/get/diva2:945707/FULLTEXT01.pdf.

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

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

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

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

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

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

[53] Johan Wilander. *OWASP Sweden: Domändriven säkerhet / Domain-Driven Security*. 2009. URL: http://owaspsweden.blogspot.se/2009/09/domandriven-sakerhet-domain-driven.html (visited on 01/25/2018).

[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

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