Dynamic Taint Tracking for Domain Driven Security (DRAFT)

FREDRIK ADOLFSSON

Master in Computer Science
Date: May 18, 2018
Supervisor: Musard Balliu
Examiner: Mads Dam
Swedish title: Dynamisk taint spårning för domändriven säkerhet
School of Computer Science and Communication

Abstract

Todo:

Sammanfattning

Todo:

Contents

Chapter 1

Introduction

The creation of the World Wide Web (web) has caused a huge impact on today's society [45]. The web is a source for information and it connects the world through a unanimous 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 for the targeted user groups but for anyone with access to the web. This entails that malicious users who wish to abuse and/or cause harm to other users have the accessibility to possibly do so.

There are several possible attacks that a web application is vulnerable to. The attack most frequently conducted today will probably not be the same as the most conducted 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 [27]. OWASP have produced reports about the top 10 security risks for a web application and the latest was published 2017. In this report was Injection Attacks number one and Cross-Site Scripting number seven [29, 27, 9].

To minimize the risk of accidentally introducing security flaws in to the application have a variety of tools and methodologies been created. One of these is Dynamic Taint Tracking which marks input from the user as tainted through a taint variable attached to data type representing the input data. This taint variable follows the input throughout the application and propagates onto the other variables it encounters. It is possible to detaint the input but this is only done after the input have been validated. The taint value is later checked in sensitive areas through something called sinks. Execution is halted if a tainted variable is detected trying to enter the sensitive area through the sink [31, 42].

One of the methodologies that have been coined is the programming paradigm Domain Driven Security. Domain Driven Security aim to secure applications by focusing on the core domain models and making certain that validation of the value objects is correct [44, 22].

The following sections of the chapter will 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 *Related Work* in the field and 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 which is constructed to solve one or more tasks for the user.

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

Definition 1.1.3. Taint denotes marking data with a flag indicating possibility to be harmful for 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 sanitized data to be safe to the system.

Definition 1.1.8. Domain is explained in Secure by Design [25] as part of the real world where something happens.

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

1.2 Problem

> *How can an 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 have been proven to secure applications from Injection and Cross-Site Scripting attacks. Is it then possible to achieve the security gains of Domain Driven Security through applying Dynamic Taint Tracking to web applications. What would the possible 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 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 Related Work

Stendahl [38] wrote a thesis in 2016 where he evaluated if a Domain Driven Security can prevent Injection Attacks and Cross-Site Scripting. He concluded 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 objects.

Haldar, Chandra, and Franz [17] have written a report about Dynamic Taint Tracking in Java where they try to solve the problem of not properly validating user input. They managed to construct a tool that is independent from the web applications source code and the results from using the tool is a 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 to be tested.

There do exist two Dynamic Taint Tracking tools where Phosphor [33] is one and Security Taint Propagation [12] is another. Both are open source projects and developed for Java applications.

1.5 Delimitations

The focus of the thesis lies on web applications security vulnerabilities. However, other application areas might be vulnerable to the same kind of vulnerabilities. Information and discussions about those areas will not be given.

Delimitations for the application is that it will only consist of a Dynamic Taint Tracker and not, in any form, a static version. The application will only be developed for Java application. Development will be conducted in Java with the help of the library Javassist.

1.6 Methodology

The methodology of this thesis is a combination of quantitative and qualitative research. The first part of the thesis consists of a literature study followed by reasoning about tainting and detainting rules. This is done through qualitative research. The Second part of the thesis is to evaluate the implemented Dynamic Taint Tracking tool. This is done quantitatively where benchmarks evaluating performance and security gain functionality will be conducted and evaluated.

Chapter 2

Background

This Chapter will present background knowledge needed to comprehend 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* and *Domain Driven Security*. The last section is a chapter about *Java*.

2.1 Web Application

To make applications available for large 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 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]. A illustration of the three layer 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 safe guard for tier three where the valuable and possibly sensitive information is stored. The storage tier contains all the information the application needs to provide the wanted service. Such information might for

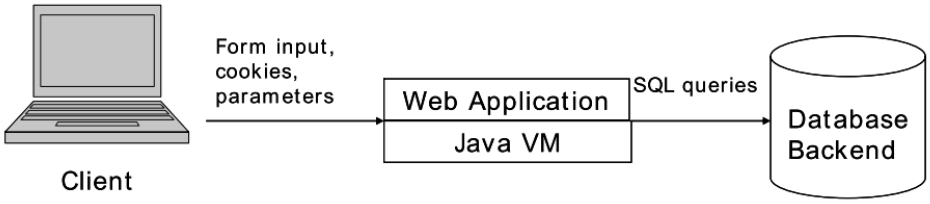

Figure 2.1: Web application architecture [17].

example be name, email, personal number and credit card information [7].

The scope of the thesis lies in tier two where the brain of the application lies. 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 standardize language called Structured Query Language, mostly known as SQL. SQL is created to programmatically manipulate and access databases. The clear majority of today's database uses SQL. The language works by building queries specifying the wanted information or task. The query will be evaluated and handled up upon by the SQL engine [11].

2.2 Security Vulnerabilities

The organization Open Web Applications Security Project, mostly known for its shortening OWASP, is an online community which aim to provide knowledge how to secure web applications [27]. OWASP have produced reports about the top 10 security risks with a web application and the latest was published 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 [29].

2.2.1 Injection Attacks

The most common security risk is Injection Attacks [29]. Injection Attack is any attack where the attacker's input changes the intent of the execution. Common result of Injection Attacks is 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 which results in gaining content or executing command on the database which was not intended. Listing 2.1 displays a SQL Query which is open to SQL Injections. This is due to the fact that the variable UserId is never validated before it is propagated into the query [7, 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, notated with *userInput*, is a valid Integer (since Integer is what we have decided that user id is in this 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 would result in query 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 input as seen in listing 2.3 can we protect the query from being assessable to SQL Injection.

Listing 2.3: Preventing SQL Injection through Verification

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

A second more 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 like SQL Injection. The only difference is that that attacker dose not receive the wanted information 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, 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, which is by analysing the error messages, and depending on what they return build an image of the wanted answer. This is mostly done by testing different combination of true and false questions [7, 39].

2.2.2 Cross-site Scripting

Cross-Site Scripting (XSS) have been a vulnerability since the beginning of the internet. One of the first XSS 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 want to gain information of. The attacker could then through JavaScript access any content that is visible or typed into the web application. To prevent this form of attack were the standard of Same-Origin Policy introduced. Same-Origin Policy restricts JavaScript to only access content from its own origin [15, 35].

The introduction of the Same-Origin Policy did not stop the attackers from preforming XSS 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 [15].

XSS can be divided into three different sub categories. Which are: reflected, Stored and DOM Based XSS.

Reflected XSS

Reflected XSS is 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 though the input reflect back content to the user [39].

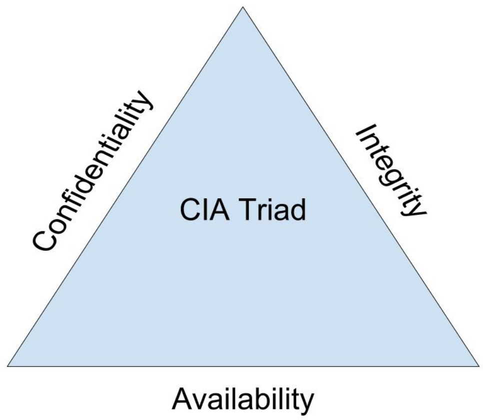

Figure 2.2: CIA Triad

Stored XSS

Stored XSS is when malicious scripts are stored in the targeted web applications database. This malicious script is then loaded and presented for each user which is trying to access the application [39].

DOM Based XSS

DOM Based XSS is very similar to Reflected XSS but it does not necessary have to be reflected from the application server. DOM Based XSS modifies the DOM tree and can exploit the user [39].

2.2.3 CIA Triad

Discussions about application security often relies on the CIA Triad which represent the three primary concepts in information security. These three are confidentiality, integrity and availability. Confidentiality are rules that specifies the access restrictions to the application. Integrity specifies that application data should be accurate and not altered. Availability is about 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 both attacks towards the confidentiality and integrity of a system. they are attacks towards confidentiality when the attackers intention is 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.

2.3 Dynamic Taint Tracking

Taint tracking, also known as taint analysis, taint checking and taint propagation, is a tool to analyse the flow of information in a domain [31]. 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 possibly harmful data. Taint tracking can be done in two different ways: static and dynamic. The static is an evaluation tool which is done statically before runtime. Dynamic Taint Tracking is a tool that is executed in runtime. The tool works by tracking data in runtime and actively blocking any data that is trying to enter the sink without being detainted through sanitation first. Perl and Ruby are two programming languages which have adapted to user Taint checking [32, 23]. There are some tools who enables taint checking for other languages such as TaintDroid [24] and FlexiTaint [42]. This thesis will handle Dynamic Taint Checking and how it can increase the security of an application.

This is done by marking untrusted input from sources, which is a marking point where malicious data might enter the system, 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 (remove the taint flag) 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 [31, 42]. The decision of what to do when a tainted variable try to pass through a sink might vary depending on the application. However, the common reaction is to stop the execution of the tainted code. Other actions such as logging or raising an alarm are also common.

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 line one the input from the source is flagged tainted and the taint propagates onto *userId*. The sanitizer on line two validates *userId* and removes the taint flag. Lastly, the sink on line three execute since the argument is not tainted. If a user sends in a malicious userId containing "101 OR 1 = 1" the validator would either halt the execution or sanitize the String. 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, would result in 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 to enter the application. There are security policies restricting input from sources to pass through sinks without first being sanitized through validation. The same application could however be used to enforce policies restricting sensitive data from sinks to pass through sources without being sanitized to not contain sensitive data.

This thesis will implement and evaluate a Dynamic Taint Tracking tool to prevent confidentiality and integrity vulnerabilities in 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.

2.4 Domain Driven Security

There exists a plethora of tools who aim to help in the process of developing complex domain models, but Domain Driven Design is not one of them [2, 21]. Domain Driven Design is more of a thought process and methodology to follow every step of the process [14]. In *Domain-driven design reference: definitions and patterns summaries* do Evans [13] 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 your product that is most important and often is your main selling point compared to other similar products [26]. A discussion and even possible a documentation describing the core domain is something that will help the development of the product. The idea is to keep everybody on the same track heading in the same direction [14].

The second idea is to explore and develop every model in collaboration between domain practitioners, who are experts in the given domain, and software developers. This ensures that important knowledge needed to successfully develop the product is communicated back and forth between the two parties [26]. The third idea is important 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 important parts which is to develop the product [13].

Evans [13] 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 great risk of misunderstandings and erroneous assumptions in the collaborations between the different models [26].

Wilander [44] and Johnsson [22] created 2009 a blog post each in a synchronous manner where they together introduces 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 Design is about developing complex domain models and one of the most basic rule of application security is to always validate input data. Domain Driven Security in other hand, is about the importance of creating and maintaining domain models who are reflecting the product correctly and that they are validated so they can't be populated with erroneous data [44, 22, 1, 38].

2.5 Java

Java have 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. [28]. Today is Java one of the most common programming languages [16].

Java is a statically typed language which means that no variable can be used before they have been declared. These variables can be of two different types. These are primitives and references to objects. Among the primitives dose Java have support for eight. These are byte, short, int, long, float, double, boolean and char [34].

2.5.1 Java Virtual Machine

There exists a plethora of implementation of the JVM but the official that Oracle develop is HotSpot [40]. 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, known as JVM. [8]. The idea behind the JVM was to have one language that executed the same on all platforms. And then modify the JVM to be able to run on as many platforms as possible. The JVM is a virtual machine with its own components of heap storage, stack and program counter, method area and runtime constant pool.

Figure 2.3 illustrates the architecture of the JVM. The compiled Java code that the developer creates is loaded through the Class Loader and added into the JVM Memory. [43].

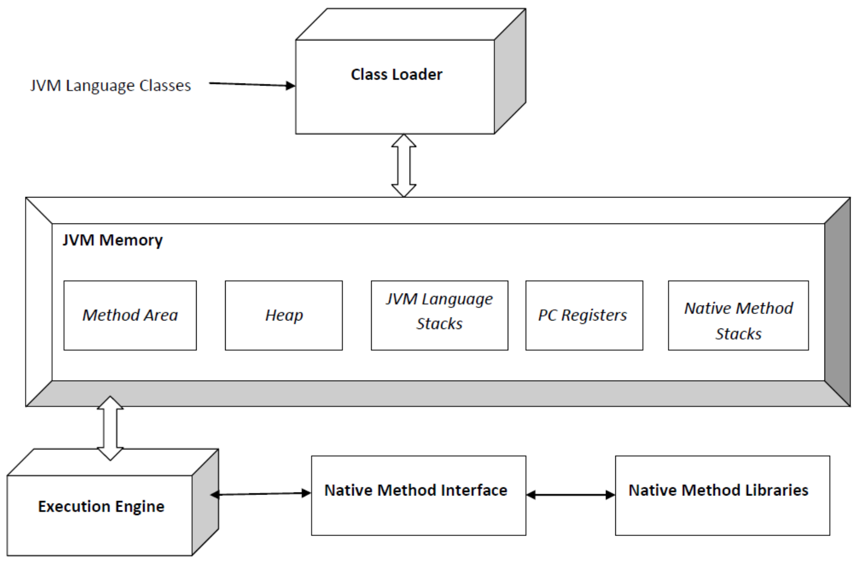

Figure 2.3: Java Virtual Machine Architecture

2.5.2 Instrumentation

Java Instrumentation is a way to modify the execution of an application on the Java Virtual Machine (JVM) without having knowledge nor the need of modifying the application code itself. This makes it beneficial to implement for example monitoring agents and event loggers through Java Instrumentation. Instrumentation is a 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 modify any application loaded in runtime [19].

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 can be translated into bytecode that the JVM 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 functionality of editing class files on source level which require no understanding of Java bytecode [20].

Chapter 3

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 chapter is then followed by *Software Architecture* and *Notable Problems*.

3.1 Policies

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

3.1.1 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.

(??? Information from sinks should not go where. Any where els than defined exits. ???)

3.1.2 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 of a sink without first being sanitized.

3.1.3 Taint Checking

The policies above will be enforced by forcing validation of data that is 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 reduces severely. One core policy preventing this is as well that no unintended code shall be able to execute.

The policies above can also be combine into 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.

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 straight forward but Instrumentation of Java primitives is a rather complex problem. However, the main behind the propagation is the same for all data types.

Below are rules defining when taint variables should propagate.

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

3.2 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 in runtime into source, sink or sanitizer.

Xboot Project that loops through all classes in rt.jar and transforms into source, sink or sanitizer.

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

3.2.1 Project Utils

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

- Is same class as defined source, sink or sanitizer.
- Implements interface of defined source, sink or sanitizer.
- Extends defined source, sink or sanitizer class (recursive call. Checks all in list for each extended class).

If a class fulfils 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 for the three different kinds. Where Instrumentation of sources will set the return parameter of the method as tainted. For sinks will a check where the taint flag is checked and if it is tainted is an error thrown. Instrumentation of sanitizers is works by detainting the return value of the method.

3.3 Notable Problems

One of the first problems that was introduced during the development phase of the application is that some classes can't be instrumented during runtime. More precisely, the classes that the JVM relies on can't be instrumented in realtime. But these is a solution to this. The solution is to create a JAR file with statically modified versions of the classes. In this case is the String class one of these. This JAR file can then be loaded through the option Xbootclasspath/p that appends the JAR file to the front of the bootstrap path. Forcing the JVM to use our modified versions of classes [18]. Because of this limitation were the decision of instrumenting as many classes as possible statically. This is to keep the code consistent.

Another problem is that primitives can't be instrumented. 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 variables that lie in the closest class or objective to the byte- or char-array. This shadow variable will contain the taint.

Possible solution to instrument primitives is to transform all primitives into their corresponding class, so call Unboxing. This logic is used in a similar manner called Autoboxing. This is used to transform the primitive, which only holds a value, into its corresponding object which contains sets of functions. Un idea I had, to solve the problem with instrumenting primitives, is to Unbox all primitives in runtime and never use primitives. This would make it easy to instrument each of the primitives corresponding classes to propagate taint. However, this is probably not a optimal solutions. The reason to this is the added overhead it would add to the execution [4].

Another problem that emerged was that operations with primitives are direct bytecode translations. Two examples of these are usage of + (addition) and - (subtraction).

Chapter 4

Evaluation

This section describes the conduction of the benchmarking and comparison of the implemented Dynamic Taint Tracker. The chapter starts with a description of the *Test Environment* followed by a detailed description about the *Benchmarking*

4.1 Test Environment

The execution of the benchmarking is conducted on a Asus Zenbook UZ32LN with the following specification:

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

Execution if the applications are executed through the command line command "java". Executions of benchmarking tests where no Dynamic Taint Tracker is enabled is run without a flag. Executions of tests with Dynamic Taint Trackers are called with the two flags "javaagent" and

”xbootclasspath/p” where the first will point towards the Agent project jar and the second points towards the Xboot project jar.

4.2 Benchmarking

batik not wokring in DaCapo

Chapter 5

Result

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

5.1 Performance Overhead

The results from benchmarking the application on DaCapo Benchmark Suit [10] is seen in Figure 5.1 and 5.2. Both graphs are constructed to show the added overhead of running the applications with Dynamic Taint Tracking activated. The graphs are conducted based on the data in Table A.1 and A.2.

5.1.1 Time

Figure 5.1 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 Fop with an overhead of 426.2%. The average overall is 142.1%.

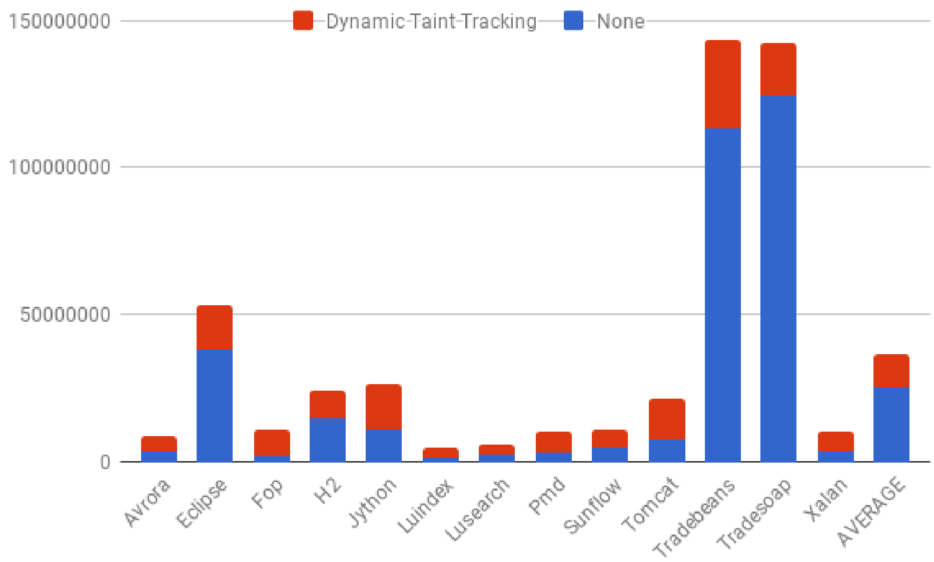

Figure 5.1: Average Added Time in Microseconds

5.1.2 Memory

Figure 5.2 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 Fop with an overhead of 277.8%. The average overall is 127.2%.

5.2 Applications

The presented results in this chapter are from evaluating Java applications for security vulnerabilities with and without Dynamic Taint Tracking. The results from each application is listed in its own table where vulnerability type and number of vulnerabilities is listed. In the presentation of the result in text are vulnerabilities of the same type aggregated.

Table 5.1 shows the vulnerabilities from evaluating Stanford SecuriBench Micro [37]. In the table can we see that the most frequent vulnerability is reflected XSS where 71 vulnerabilities are present. Second most

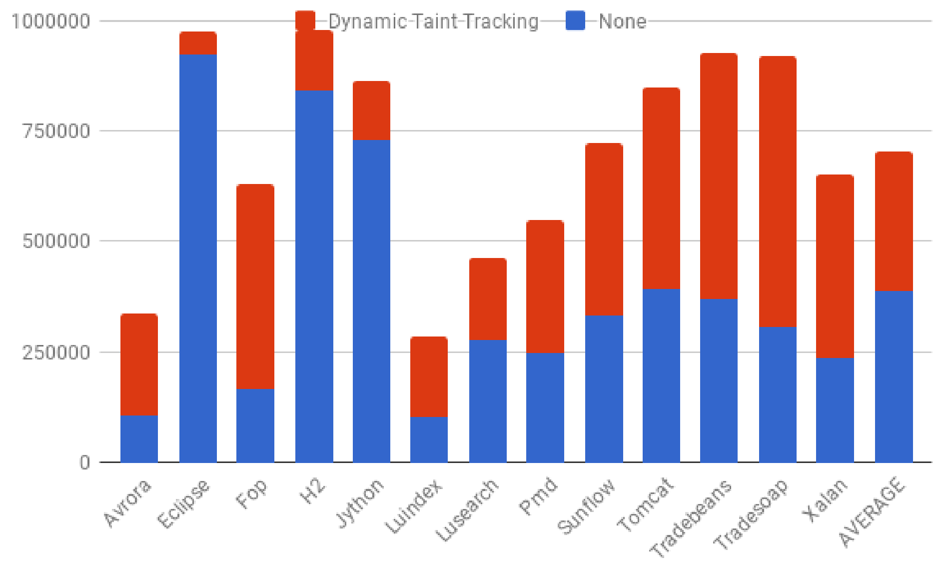

Figure 5.2: Average Added Memory in Kilobytes

common is SQL Injection with 20 and the least common with 1 vulnerability is Buffer Overflow. By enabling Dynamic Taint Tracking on the Stanford SecuriBench Micro [37] application results in a 100% prevention rate.

Table 5.1: Security Vulnerabilities Found in Stanford SecuriBench Micro

	None	**Dynamic Taint Tracking**
Cross Site Scripting (Reflected)	71	0
SQL Injection	20	0
Buffer Overflow	1	0

Table 5.2 shows the vulnerabilities from running Insecure [30] with and without Dynamic Taint Tracker. Of the two types of vulnerabilities is SQL Injection the first with six vulnerabilities abd reflected XSS with two. Enabling Dynamic Taint Tracking on Insecure [30] results in 100% prevention rate on SQL Injection attacks and 0% for XSS. The overall prevention rate is 75%.

The results from evaluating the application SnipSnap [36] is seen in Table 5.3. In this table can we see that the most common vulnerability is reflected XSS with 172 occurrences. Second Largest is SQL Injection

Table 5.2: Security Vulnerabilities Found in Insecure

	None	Dynamic Taint Tracking
Cross Site Scripting (Reflected)	2	2
SQL Injection - Authentication Bypass	2	0
SQL Injection - Hypersonic SQL	4	0

with 49 occurrences followed by CRLF Injection with two. Enabling Dynamic Taint Tracking yields in a overall prevention rate of 77.2%. All CRLF Injection are prevented. XSS is prevented to 77.3% and SQL Injection with 75.5%.

Table 5.3: Security Vulnerabilities Found in SnipSnap

	None	Dynamic Taint Tracking
Cross Site Scripting (Reflected)	172	39
CRLF Injection	3	0
SQL Injection	47	10
SQL Injection - Authentication Bypass	2	2

Table 5.4 shows the vulnerabilities from evaluating Ticketbook [41]. The most common vulnerability was XSS with 14 occurrences. SQL Injection was the least with one. The prevention rate of SQL Injection was 100% and for XSS 71.4%. The overall prevention rate is 73.3%.

Table 5.4: Security Vulnerabilities Found in Ticketbook

	None	Dynamic Taint Tracking
Cross Site Scripting (Persistent)	2	0
Cross Site Scripting (Reflected)	12	4
SQL Injection	1	0

Chapter 6

Discussion

Todo: Results show that memory overhead is large in some areas. this could be optimized and probably lowered. But this is not a good solution for memory sensitive domains.
The largest added overhead commes from the instrummentation of application during rutime. Meaning that first time a class is imported will be slower but the second will be almost as fast as None. This is proben by treadbeans and tradesoap. This could be solved by instrummenting the application beforehand.

6.1 Domain Driven Security

Todo: DDS is proven in earlyer report to combat Injection and XSS. Results prove that number of attacks are lowered by DTT. If only validating in domain primitives in DDS will missing using dp be catastrofal. DTT will still notify missing validation.
To enforce usage of DDS could sanitation of datatypes only be conducted in constructors. This would force the user to only use domain primitives.

6.2 Sources, Sinks and Sanitizers

Todo: Discuss the complexity with declaring SSS. How Should this be solved? Subscribe to list depending on used libraries.

6.3 Methodology of Evaluation

Todo: Compare to similar applications. Phosphor dose not support detainting. Meaning not applicable as Dynamic Taint Tracker for applications in production where the goal is to halt the execution of taint exceptions.

Chapter 7

Future Work

Todo: Optimize code (Memory and Time), Extend datatypes supporting propagation, Idealy implement support for multiple list subscription (source, sink, sanitizer lists), Implement support for larger set of taintflags (sanitizers can detaint for sertain type of sinks, or all)
DDS not fully established and still under ”construction”. Meaning that further evaluations comparing executions of applications fully developed with the methodology DDS.

Chapter 8

Conclusion

Todo:

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] Steven C Bankes. "Tools and techniques for developing policies for complex and uncertain systems Introduction: The Need for New Tools". In: (). URL: http://www.pnas.org/content/99/suppl%7B%5C_%7D3/7263.full.pdf.

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

[4] Joshua Bloch. *Effective Java*. eng. 2. ed.. The Java series. Upper Saddle River, NJ ; Harlow: Addison-Wesley, 2008. ISBN: 0-321-35668-3.

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

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

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

[13] Eric Evans. *Domain-driven design reference: definitions and patterns summaries*. Dog Ear Publishing, 2015.
[14] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. eng. Boston, Mass.: Addison-Wesley, 2004. ISBN: 0-321-12521-5.
[15] Seth Fogie. *XSS attacks cross-site scripting exploits and defense*. eng. Burlington, MA: Syngress, 2007. ISBN: 1-281-06024-0.
[16] *GitHub Octoverse 2017 | Highlights from the last twelve months*. URL: `https://octoverse.github.com/%7B%5C#%7Dwork` (visited on 03/21/2018).
[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] *Javassist by jboss-javassist*. URL: `http://jboss-javassist.github.io/javassist/` (visited on 02/27/2018).
[21] Rabia Jilani et al. "ASCoL: A Tool for Improving Automatic Planning Domain Model Acquisition". In: *AI*IA 2015 Advances in Artificial Intelligence*. Ed. by Marco Gavanelli, Evelina Lamma, and Fabrizio Riguzzi. Cham: Springer International Publishing, 2015, pp. 438–451. ISBN: 978-3-319-24309-2.
[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] FIX ME!!! *Secure by Design*. FIX ME!!!, 2018.

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

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

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

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

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

[31] 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.

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

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

[34] *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).

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

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

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

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

[42] 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.

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

[44] 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).

[45] *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

Todo: Update all tables with final result and introduce with short summary

Table A.1: Time Overhead (ms)

	None			Dynamic Taint Tracking		
	Average	**Min**	**Max**	**Average**	**Min**	**Max**
Avrora	3813025	3744824	3866363	9042154	8325428	9523650
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)

	None			**Dynamic Taint Tracking**		
	Average	**Min**	**Max**	**Average**	**Min**	**Max**
Avrora	108445	99716	122236	336260	240968	407668
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