Integrating Static Taint Analysis in an Iterative Software Development Life Cycle

Thomas Lie and Pål Ellingsen
Department of Computing, Mathematics and Physics
Western Norway University of Applied Sciences
Bergen, Norway
Email: thomas.lie@student.hib.no, pal.ellingsen@hvl.no

***Abstract*—Web applications expose their host systems to the end-user. The nature of this exposure makes all Web applications susceptible to security vulnerabilities in various ways. Two of the top problems are information flow based, namely injection and cross-site scripting. A way to detect information flow based security flaws is by performing static taint analysis. The idea is that variables that can directly or indirectly be modified by the user are identified as tainted. If a tainted variable is used to execute a critical command, a potential security flaw is detected. In this paper, we study how to integrate static taint analysis in an iterative and incremental development process to detect information flow based security vulnerabilities.**

Keywords–Taint Analysis; iterative development; software security; injection attacks.

I. INTRODUCTION

The Open Web Application Security Project (OWASP) analyses data from software security firms and periodically publishes a report about the top 10 most common security vulnerabilities found in Web applications. The data analysed covers over 500,000 vulnerabilities over thousands of applications making this list a well documented ranking of the most common vulnerabilities present in Web applications today [1]. Two of the types of vulnerabilities at the top of the OWASP top 10 list are information flow based, namely injection and cross-site scripting. Being information flow based means that in order for an attacker to successfully exploit the type of vulnerability, untrusted data must enter the application. This untrusted data then bypasses the validation due to a poor validation routine or a complete lack of validation. When the untrusted data eventually reaches the critical command the attacker aimed for, the vulnerability is exploited. In the category of injection based vulnerabilities reside numerous exploitable implementations, such as queries for SQL (Structured Query Language), LDAP (Lightweight Directory Access Protocol), Xpath, NoSQL and command injection in the form of operating system commands or program arguments. Due to the widespread use of database access based on SQL in Web applications, the most common injection vulnerability is therefore SQL injection. Two other types of information flow vulnerabilities that are worth briefly mentioning are path traversal and HTTP (Hypertext Transfer Protocol) response splitting. Path traversal allows an attacker to access or control files that are not intended by the application. This can happen if the application fails to restrict access to the file system. Path traversal belongs in the category of insecure direct object references in the OWASP top 10 [1] [2].

Numerous approaches for detecting SQL injection and cross-site scripting are documented. Some of them are briefly described in the following paragraphs. *SQLUnitGen* is a tool to detect SQL injection vulnerabilities in Java applications. First, the tool traces input values that are used for an SQL query. Based on this analysis, test cases are generated in the form of unit tests with attack input. Lastly, the test cases are executed and a test result summary showing vulnerable code locations is provided [3]. Fine-grained access control is more of a way of eliminating the possibility for SQL injection rather than detecting it. The concept is to restrict database access to information only the authenticated user is allowed to view. This is done by assigning a key to the user, which is required in order to successfully query the database. Access control is in fact moved from the application layer to the database layer. Any attempt to execute SQL injection cannot affect the data of different users [4].

SQLCHECKER is a runtime checking algorithm implementation for preventing SQL injection. It checks whether an SQL query matches the established query grammar rules, and the policy specifying permitted syntactic forms in regards to the external input used in the query. This means that any external input is not allowed to modify the syntactic structure of the SQL query. Meta-characters are applied to external input functioning as a secret key, for identifying which data originated externally [5]. Brower-enforced embedded policies is a method for preventing cross-site scripting vulnerabilities. The concept is to include policies about which scripts are safe to run in the Web application. Two types of policies are supported. A whitelisting policy is provided by the Web application as a list of valid hashes of safe scripts. Whenever a script is detected in the browser, it is passed to a hook function hashing it with a one-way hashing algorithm. Any script whose hash is not in the provided list is rejected [6]. The second policy, Document Object Model (DOM) sandboxing, is used to enable the use of unknown scripts. This could be a necessary evil for a Web site that, for example, requires scripts in third-party ads. Contrary to the first policy, this is a blacklisting policy. The Web page structure is mapped, and any occurrences of the noexecute keyword within a or element enables sandbox mode in that element, disallowing running scripts [6]. The methods covered in the preceding paragraphs for both detecting and/or preventing SQL injection and cross-site scripting have one thing in common. All approaches present detection solutions limited to their respective vulnerability, being it either SQL injection or cross-site scripting. Since both types of vulnerabilities belong to the same category of vulnerabilities, information flow vulner-

 ISBN: 978-1-61208-553-1

abilities, a mutual approach is desirable to explore. Such an approach should also be able to detect all forms of information flow vulnerabilities. *FindBugs* [7] is a popular static analysis tool for Java. It has a plugin architecture allowing convenient adding of bug detectors presently detecting both SQL injection and cross-site scripting. The bug detectors analyse the Java bytecode in order to detect occurrences of bug patterns. Up to 50% false warnings may be acceptable if the goal of the analysis is just to get a general idea of where to do coding improvements in a development process. Having a much more precise analysis reporting none or low false warnings saves the developers time. Therefore, finding a method with a much higher accuracy is preferable. The approach that is explored in this paper in order to detect information flow vulnerabilities, is the approach called taint analysis.

In the following, we want to study how taint analysis can be integrated in the development process, and how suitable the existing implementations are for this kind of integration. To carry out this study, we have applied the analysis to the development of a Java Enterprise Edition (Java EE) application throughout the development process. The outline of the rest of this paper is as follows. Section II describes the principles of taint analysis, and some implementations of this technique. In Section III, the methodology used in this study is presented. Based on this, the results and an analysis of these is presented in Section V. Finally, our findings are summed up in Section VI.

II. Taint Analysis

Taint analysis resides within the domain of information flow analyses. Essentially, this means that tracking how variables propagate throughout the application of analysis is the core idea. In order to detect information flow vulnerabilities, entry points for external inputs in the application need to be identified. The external inputs could be data from any source outside the application that is not trusted. In other words, it must be determined where there is a crossing in the applications established trust boundary. In a Web application context, this is typically user input fetched from a Web page form, but would also include, e.g., URL parameters, HTTP header data and cookies. In taint analysis, the identified entry points are called sources. The sources are marked as tainted, and the analysis tracks how these tainted variables propagate throughout the application. A tainted variable rarely exclusively resides in the original assigned variable, and thus it propagates. This means that it affects variables other than its original assignment. This can happen directly or indirectly. Directly in that, e.g., a tainted string object is assigned either fully or partly to a new object of some sort. An example of indirect propagation is when a tainted variable that contains an id is used to determine what data is assigned to a new variable, see Figure 1 [8].
A tainted variable in itself is not harmful to an application. It is

```
1 HashMap map = ...;
2 String id = request.getParameter("id"); //Source
3 User user = (User) map.get(id);
```

Figure 1: A tainted source variable containing an id to fetch data from a HashMap indirectly induces taint on an object.

when a tainted variable is used in a critical operation without proper sanitization, that vulnerabilities could be introduced. Sanitizing a variable means to remove data or format it in such a way that it will not contain any data that could exploit the critical command in which it will be used. An example is when querying a database with a tainted string, it could open for SQL injection if the string contains characters that either change the intended query, or split it into additional new queries. Proper sanitization would remove the unwanted characters, eliminating the possibility of unintended queries and essentially preventing SQL injection. Contrary to input data being assigned as sources, methods that executes critical operations are called sinks in taint analysis. When a tainted variable has the possibility to be used within a sink, a successful taint analysis implementation would detect this as a vulnerability. Taint analysis can be divided into two approaches, dynamic taint analysis and static taint analysis. The dynamic taint analysis approach analyses the different executed paths in an application specific runtime environment. Tracking the information flow between identified source memory addresses and sink memory addresses is generally how this kind of analysis is carried out. A potential vulnerability is detected if an information flow between a source memory address and a sink memory address is detected. Static taint analysis is a method that analyses the application source code. This means that, ultimately, all possible execution paths can be covered in this type of analysis, whereas in a dynamic taint analysis context, only those paths specifically included in the analysis are covered.

Dynamic taint analysis can be used in test case generation to automatically generate input to test applications. This is suitable for detecting how the behaviour of an application changes with different types of input. Such an analysis could be desirable as a step in the development testing phase of a deployed application since this could also detect vulnerabilities that are implementation specific. Dynamic taint analysis can also be used as a malware analysis in revealing how information flows through a malicious software binary [9]. Taking this analysis one step further enables malicious software detection of, e.g., keyloggers, packet sniffers and stealth backdoors. The concept is to mark input from keyboard, network interface and hard disk tainted, and then track the taint propagation to generate a taint graph. By using the taint graph in automatically generating policies through profiling on a malicious software free system, detection of anomalies is possible. E.g., in the case of detecting keyloggers, the profile includes which modules would normally access the keyboard input on a per application basis. When a keylogger is trying to access a specific profiled application, this could be detected [10]. In both static and dynamic taint analysis implementations, the precision of the analysis is important for it to be trustworthy. Generally, two outcomes can affect the analysis precision. The first scenario is when the analysis for some reason marks a variable as tainted that has not propagated from a tainted variable. This is called over tainting and leads to false positives, which means that the reported error is not truly an error. The second outcome is when the analysis misses an information flow from a source to a sink. Thus, the analysis does not report an error that actually is present. This is called under tainting, and the term false negative describes the absence of an actual error [9]. Dynamic taint analysis has, as shown in previous paragraphs, several

 ISBN: 978-1-61208-553-1

types of applications. However, static taint analysis may be a better fit for integration within the development process due to the direct analysis of source code. There are different ways to implement static taint analysis, and we have considered three different implementations for Java, which are elaborated in the following.

A. Implementations of taint analysis

An implementation of taint analysis for Java, described by Tripp et al. [11], consists of two analysis phases. The first phase performs a pointer analysis and builds a call graph. Pointer analysis, also called points-to analysis, enables mapping of what objects a variable can point to. A call graph in this context is static, which means that it is an approximation of every possible way to run the program in regards to invoking methods. Tripp et al. describe an implementation of specific algorithms, but the analysis design is flexible in that using any set of desired algorithms is feasible [11]. The second phase takes the results of the first phase as input and uses a hybrid thin slicing algorithm to track tainted information flow. Thin slicing is a method to find all the relevant statements affecting the point of interest, which is called the seed. In comparison to a traditional program slicing algorithm, thin slicing is lightweight in that it only includes the statements producing the value at the seed. This means that the statements that explain why producers affect the seed are excluded in a thin slice [12]. Thin slicing works well with taint analysis because the statements most relevant to a tainted flow are captured. Hybrid thin slicing essentially produces a Hybrid System Dependence Graph (HSDG) consisting of nodes corresponding to load, call and store statements. The call statements represent source and sink methods. The HSDG has two types of edges, direct edges and summary edges, that represent data dependence. The data dependence information is computed in the first phase by pointer analysis. Tainted flows are found by computing reachability in the HSDG from each source call statement, adding the necessary data dependence edges on demand [11]. The way this implementation defines sources and sinks is through security rules. Security rules exist in the form $(S1, S2, S3)$. $S1$ is a set of sources. A source is a method having a return value which is considered tainted. $S2$ is a set of sanitizers. A sanitizer is a method that takes a tainted input as parameter and returns that parameter in a taint-free form. $S3$ is a set of sinks. Each sink is defined as a pair (m, P), where m is the method performing the security sensitive operation and P defines the parameters in m that are vulnerable when assigned with tainted data [11]. This implementation of taint analysis for Java includes ways to incorporate Web application frameworks in the analysis. External configuration files often define how the inner workings of a framework is laid out. Therefore, a conservative approximation of possible behaviour is modelled. For the Apache Struts framework, which is an implementation of the Model View Controller (MVC) pattern, the Action and Action Form classes are treated as sources. These classes contain execute methods taking an ActionForm instance as a parameter. This instance contains fields which are populated by the framework based on user input meaning it should be considered tainted. Thus, the analysis implements a model treating the Action classes as entry points.

An alternative static taint analysis implementation is similar to Taint Analysis for Java in that it is based on pointer analysis and construction of a call graph. However, this implementation depends on pointer analysis and call graph alone in detecting tainted flows. The analysis uses binary decision diagrams in the form of a tool called *bddbddb* (BDD-Based Deductive DataBase), which includes pointer analysis and a call graph representation [2]. Binary decision diagrams can be utilized in adding compression to a standard binary decision tree based on reduction rules. In the context of this analysis, the compression of the representation of all paths in the call graph makes it possible to efficiently represent as many as 10 contexts. This allows the analysis implementation to scale to applications consisting of almost 1000 classes [2]. In order to detect vulnerabilities, specific vulnerability patterns need to be expressed by the user. A pattern consists of source descriptors, sink descriptors and derivation descriptors. Source descriptors specify where user input enters the application, e.g., `HttpServletRequest.getParameter(String)`. Sink descriptors specify a critical command that can be executed, e.g., `Connection.executeQuery(String)`. Lastly, derivation descriptors specify how an object can propagate within the application, e.g., through construction of strings with `StringBuffer.append(String)` [2]. Tainted Object Propagation Analysis does not implement any handling of Web application frameworks.

A third implementation, Type-based Taint Analysis, differs from the preceding approaches in that a type system is the basis of the analysis. The implemented type system is called SFlow, which is a context-sensitive type system for secure information flow. SFlow has two basic type qualifiers, namely tainted and safe. Sources and sinks are identified in these methods, and fields are annotated using these type qualifiers. A type system is a system that intends to prove that no type error can occur based on the rules established. This is done by assigning a type to each computed value in the type system, and the flow of these values is then examined. This concept is called subtyping [8]. The subtyping hierarchy is defined as *safe* $<:$ *tainted*. This means that a flow from tainted sources to safe sinks is disallowed. The other way around, assigning a safe variable to a tainted variable is allowed. A third type of qualifier, *poly*, is included in order to correctly propagate tainted and safe variables through object manipulation, e.g., with `String` methods `append` and `toString`. All object manipulation methods, such as `String append` and `toString`, would be annotated as $poly$. The $poly$ qualifier in combination with viewpoint adaptation rules ensures that the implementation is context-sensitive. This means that parameters returned from such methods inherit the manipulated inbound parameters type qualifier (tainted or safe). As a result, the subtyping hierarchy becomes *safe* $<:$ *poly* $<:$ *tainted* [8]. Another benefit with the *poly* qualifier implementation is that tainted variables properly propagate in third-party libraries. As a result all application code is included in the analysis. Type-based Taint Analysis also supports Web application frameworks in the same way as the regular Java API is supported, namely by annotating the relevant fields and methods. An example of this is that for the Apache Struts framework, the Action class containing the execute method is what needs to be annotated. This method takes an ActionForm instance as a parameter, that contains fields which are populated by the framework based on tainted user input. Simply annotating the ActionForm parameter as

 ISBN: 978-1-61208-553-1

Figure 2: The Software Development Life Cycle [13].

tainted would include the framework in the analysis [8]. Type inference implies identifying a valid typing based on the subtyping rules defined in the SFlow type system. A succeeded inference means that there are no flows from sources to sinks. If the type inference fails, a type error is evident meaning that a flow from a tainted source to a safe sink is present.

III. Methodology

When developing software, a common approach is to establish a Software Development Life Cycle (SDLC). The SDLCs function is to cover all processes associated with the software developed. Different types of SDLC models exist. However, whether it being Waterfall, Agile or some other model, the processes in the SDLC can be partitioned into different phases. In this paper, the phases are named according to Merkow and Raghavan [13] as Requirements, Design, Development, Test and Deployment, see Figure 2. Developing software requires planning of both functional requirements and non-functional requirements in order to deliver an acceptable end product. The functional requirements refer to the functionality of the software, whereas non-functional requirements refer to quality attributes, e.g., capacity, efficiency, performance, privacy and security. The Requirements phase addresses the gathering and analysis of requirements regarding the environment in which the software is going to operate. Non-functional requirements based on security policies and standards, and other relevant industry standards that affect the type of software developed, are included in this phase. The Design phase is where the functional requirements of the software developed are planned, based on the mapping of requirements in the first phase. This phase also includes architectural choices that determine the technologies used in the development of the software. The Development phase contains the actual coding of the software developed. Both functional requirements and non-functional requirements from the earlier planning phases are being addressed. A common approach is to develop the functional requirements in small programs called units. These units are then tested for their functionality, a methodology called Unit Testing. The Test phase is where test cases are built, based on requirements criteria from earlier phases. Both test cases for functional requirements and non-functional requirements are included. The test phase is iterative in nature meaning that the problems found would need to be addressed and fixed in the development phase. After the problems are fixed, the system would need to go through the test phase once again. The deployment phase is the final phase in the cycle, and the main activity is to install the software and make it ready to run in its intended environment, or released into the market. At this point, both testing of functional requirements and non-functional requirements are finished [13].

The problem description (see Section I) states that we will study how to integrate static taint analysis in the development process of a Java EE Web application. Given the tools proposed in Section I for detecting information flow vulnerabilities, static taint analysis is explored in this experiment. This choice is based on the fact that this type of analysis embraces the detection of the whole domain of information flow vulnerabilities. Static taint analysis may also have significantly fewer false warnings compared to e.g., analyses depending on code patterns such as the FindBugs static analysis tool. The research approach regarding the problem description is to carry out a case study in two main parts. The first part is to develop a prototype Java EE Web application of an acceptable size so that it is not too small with regard to performing taint analysis on it. This means that the prototype application should preferably have multiple modules interacting with external processes, i.e., at a minimum implementing a database connection. Further, the user interaction would naturally be done through a website utilizing specific Java EE technologies. The goal of the last part in the case study is to architect a solution to the taint analysis integration. Many aspects regarding this integration would need to be clarified. Based on the experiences with the implementation of taint analysis in the specific prototype application, general conclusions regarding the problem description can be drawn. Some important approaches to implementing static taint analysis for Java are given in [2], [8], [10] and [14]. From these approaches, summarized in Section II-A, the Type-based Taint Analysis from [8] was selected. This choice was convenient in that the analysis platform is available as an open source project and Type-based Taint Analysis also looks promising with regard to how Web application frameworks are handled. Analysing frameworks are especially relevant in Java EE Web applications, e.g., in the form of the Java Server Faces (JSF) framework managing the applications front-end. Based on how this analysis method is described in [8], it would seem that the implementation is feasible as an integrated step in a Java EE Web application development context.

IV. Integrating Taint Analysis in the SDLC

Considering that modern development practices are team based, and in fact multi-team based on big projects, it is important to include this observation in assessing whether static taint analysis can efficiently be integrated in the SDLC. An agile development methodology including an iterative and incremental workflow leads to developing a piece of software in numerous modules. Being able to properly test both a single module and a set of modules for detecting information flow vulnerabilities is preferable. According to Huang et al. [8], the taint analysis implementation is modular, meaning that a whole program is not necessary for analysis. This is promising considering the modern development practice described in the previous paragraph. Additionally, the taint analysis implementation should be included in the development phase along with other testing activities (see Section III) describing the different phases in the SDLC [8]. In addition to the development phase, the testing phase could include static taint analysis. However, the reason to avoid integration within

 ISBN: 978-1-61208-553-1

the testing phase is that anything added to that phase adds unnecessary overhead. Even if the overhead of running the analysis is eliminated by making it fully automated, a system for countering the output in form of requested fixes for the next development phase iteration needs some resources. Also, a known concept is that the earlier vulnerabilities are found in the SDLC, the cheaper it is to get them fixed. The aim is therefore to craft a solution to integrate static taint analysis into the development phase. Some methods for detecting and/or preventing information flow vulnerabilities are listed in Section I. Most of these methods focus exclusively on either SQL injection or cross-site scripting rendering detection of other information flow attacks uncovered. Although FindBugs is an example of a static analysis covering most, if not all, the information flow vulnerabilities, its detecting algorithm is prone to have a high percentage of false positives. The choice of type-based taint analysis in the form of SFlow was done because it can detect a high number of vulnerabilities and also has a low number of false positives.

For the case study, a Java EE based Web application for remotely controlling an automated production system was developed. The size of the project was determined to be sufficiently large to do a realistic study on the integration of static taint analysis in the development process, while at the same time being sufficiently small to focus on the research question at hand. The development resources for the case study application amounted to one developer limited to roughly three months development time. As in one man team, a natural SDLC approach to adopt is the Big Bang Model. This is simply a term made to cover an SDLC, which contains no or little planning and does not follow any specific processes [15]. Although a complete SDLC methodology was not followed during the case study project, several key activities were integrated in the SDLC in order to ensure delivery of an acceptable end product. Enabling development of the prototype application iteratively and incrementally was done by applying continuous delivery. This means that the functionality was split up and developed in smaller tasks and delivered in predefined iteration cycles of, e.g., two weeks. The prototype application was developed in iterations with an integrated static taint analysis as a part of the SDLC. While the prototype application has a limited size with a moderate number of iterations of development, we consider taint analysis conducted during the development cycle to be adequate in order to draw conclusions. The bigger the application the more value of frequent analysis. This is because the issues found earlier in a big application environment would contribute knowledge to prevent making the same mistakes over and over as the application progresses, thus saving developer resources.

V. Analysis

A main challenge during the implementation was to properly annotate external libraries, e.g., frameworks, in order to enable a working analysis without developer intervention. Adding annotations manually was not an option because, in addition to creating extra work for the developer, it is prone to errors. For SFlow to be a successful security analysis tool, we found that the annotation process needs to be improved. One approach in changing the annotation process could be to use a strategy from the paper by Sridharan et al. [14]. This paper describes a framework as a solution for adding Web application frameworks to a taint analysis implementation. In a similar way, a framework for adding annotations to the SFlow annotated Java Development Kit (JDK) could be developed easing the work of figuring out how to conduct the process of annotation. This framework could also include verification routines for testing that the annotations are working correctly [14]. Another change SFlow must undergo is the way the analysis is conducted. In its current form, SFlow exists as a manual command-line tool. For this tool to exist in the development phase without unnecessary overhead, an automatic integration of the analysis is required. Therefore, integrating SFlow as a plugin in an IDE (Integrated Development Environment) by utilizing this support by The Checker Framework could be a good solution. This would make the taint analysis convenient and seamless for the developer enabling analysis whenever the developer builds the application and/or desires to run it. However, deciding if the integration is not creating too much overhead for the developer boils down to the running time of the taint analysis implementation. Results from Huang et al. [8] state that analysing 13 relatively large applications resulted in running times of less than four minutes for all applications except one. The analysis ran on a server with Intel Xeon X3460 2.8GHz processor and 8GB RAM. As for the smaller prototype application, the running time is about 30 seconds on a laptop with Intel Core i5-3210M 2.5GHz processor and 6GB RAM [8]. Even though the running time of the taint analysis is done within minutes and may not introduce a significant overhead for the developer running the analysis in the background, implementation in a different way could be advantageous. This solution is to incorporate taint analysis in a continuous integration tool, e.g., Jenkins, by integrating SFlow in the build system it uses, e.g., Maven. By doing this, the taint analysis will automatically run on every build. The errors will then show up as compiler errors and warnings in the continuous integration tool for the developers to address. SFlow needs to undergo at least two significant changes in order to become a powerful taint analysis security tool for integration in the development phase in the SDLC. First, the annotation process for adding Web application frameworks and external libraries must become more user-friendly in order to be practical. As suggested, a solution to this would be to develop a framework for easing the annotation process. And secondly, the analysis should be integrated either in the developers development environment, or preferably within the build system of the continuous integration tool.

VI. Conclusion

Information flow vulnerabilities can occur when applications handle untrusted data. SQL injection and cross-site scripting are the most common information flow vulnerabilities. There are numerous methods presented in countering these vulnerabilities. One method, static taint analysis, looks promising in that it has the ability to cover detection of all kinds of information flow vulnerabilities. Out of three static taint analysis implementations presented in this paper, Type-based taint analysis was chosen as the preferred implementation. This approach looked promising in the way Web application frameworks are handled. The implementation is also freely available as an open-source project. A proposed solution in integrating this taint analysis approach in an iterative and incremental development process was presented. The

 ISBN: 978-1-61208-553-1

proposed solution used the developed prototype application as a manageable sized concept application for implementing taint analysis. Annotations of sources and sinks are needed to detect information flow vulnerabilities. Some libraries are already annotated in the taint analysis implementation, referred to as the annotated JDK. To properly analyse an application, all libraries containing sources and sinks in a developed application need to be included in the annotated JDK. The development of the prototype application gave a good technical understanding of the inner workings of the application. This was advantageous in order to identify what needed to be annotated. The approach of mapping the attack surface of the prototype application turned out to be an effective way to identify the libraries containing sources and sinks.

Preparing the taint analysis implementation for analysis is mostly about making sure the libraries that are used are included in the annotated JDK and are also working properly. The experiences with annotation indicates that this is not a straight forward process, and could need many resources in order to get it right. A framework for easing the process of annotation, including verification that the annotation works correctly, is proposed as a solution to this challenge. Multiple approaches to conducting the taint analysis are possible. Running the taint analysis manually in command line, integrating it in the developers IDE and integrating it in the continuous integration tool are all possibilities. The latter suggestion is proposed as the most effective solution; implementing taint analysis in the continuous integration tools build system. This is considered an effective approach because an analysis could take several minutes to complete depending on application size. Also, processes done automatically and by an external instance will not be a distraction for the developer. When to counter any detected type errors is then up to when the developer monitors the notifications given in the continuous integration tool.

VII. Further Work

In order to support static taint analysis during the development process, the next step would be to get the annotations of the application's classes to work properly. A course worth researching, as suggested, could be to develop a framework for easing the process of annotating. Further work could also include more research in the area of how to best integrate taint analysis in a development process. The proposed solution of integrating the analysis in a continuous integration tools build system is probably worth exploring. An actual proof-of-concept implementation could be using Jenkins continuous integration tool with the Maven build system.

References

[1] OWASP Foundation, "OWASP top 10 - 2013: The ten most critical web application security risks," 2013, Accessed: 2017-04-13. [Online]. Available: https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf

[2] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis." in Usenix Proceedings of the 14th Conference on USENIX Security Symposium, vol. 2013, 2005, pp. 271–286.

[3] Y. Shin, L. Williams, and T. Xie, "Sqlunitgen: Test case generation for sql injection detection," North Carolina State University, Raleigh Technical report, NCSU CSC TR, vol. 21, 2006, p. 2006.

[4] A. Roichman and E. Gudes, "Fine-grained access control to web databases," in Proceedings of the 12th ACM symposium on Access control models and technologies. ACM, 2007, pp. 31–40.

[5] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in ACM SIGPLAN Notices, vol. 41, no. 1. ACM, 2006, pp. 372–382.

[6] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in Proceedings of the 16th international conference on World Wide Web. ACM, 2007, pp. 601–610.

[7] The FindBugs Project, "Findbugs," 2015, Accessed: 2017-04-13. [Online]. Available: http://findbugs.sourceforge.net/

[8] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in International Conference on Fundamental Approaches to Software Engineering. Springer, 2014, pp. 140–154.

[9] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in 2010 IEEE Symposium on Security and Privacy. IEEE, 2010, pp. 317–331.

[10] H. Yin and D. Song, "Whole-system fine-grained taint analysis for automatic malware detection and analysis," 2007, Accessed: 2017-04-13. [Online]. Available: http://bitblaze.cs.berkeley.edu/papers/malware-detect.pdf

[11] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in ACM Sigplan Notices, vol. 44, no. 6. ACM, 2009, pp. 87–97.

[12] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," ACM SIGPLAN Notices, vol. 42, no. 6, 2007, pp. 112–122.

[13] M. S. Merkow and L. Raghavan, Secure and Resilient Software Development. CRC Press, 2010.

[14] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4F: taint analysis of framework-based web applications," ACM SIGPLAN Notices, vol. 46, no. 10, 2011, pp. 1053–1068.

[15] T. Bhuvaneswari and S. Prabaharan, "A survey on software development life cycle models," Journal of Computer Science and Information Technology, Vol2 (5), 2013, pp. 263–265.

 ISBN: 978-1-61208-553-1