International Journal on Advances in Software, vol 10 no 3 & 4, year 2017, http://www.iariajournals.org/software/

Applying Information Flow Tracking to the Development 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*—Information flow vulnerabilities such as Structured Query Language (SQL) Injection and Cross-Site Scripting are highly relevant issues in web applications. This article expands on a an earlier paper by the authors to investigate how to apply information flow tracking in the form of taint analysis to detect this domain of vulnerabilities in. Different types of taint analysis implementations exists and a challenge is how web application frameworks are handled by the taint analysis implementation. This technique is tested by a developing a prototype application for a company covering a genuine need. This application also functions as an artefact application in conducting taint analysis. Using this artefact, a proposed solution for integrating taint analysis in the process of developing Java EE web applications is tested. Analysing the results, it is shown that it is possible to integrate taint analysis in the development cycle, but it is also made clear that the technique must be improved to properly support its use in an automated build system.**

***Keywords*–Information flow tracking; taint analysis; iterative development; software security; injection attacks.**

I. Introduction

A. Background

Web applications exposes its host system to the end-user. The nature of this exposure makes all web applications susceptible to security vulnerabilities in various ways. The Open Web Application Security Project (OWASP) periodically publishes a report that covers the ten most common security problems. Two of the top problems are information flow based, namely *Injection* and *Cross-Site Scripting*. Being information flow based means that untrusted data enters the application to eventually be executed as a part of a critical command. An example of a common injection vulnerability, SQL injection, is shown in Figure 1 as a snippet from a Java servlet [1].

```
1 String username = request.getParameter("username");
2 String password = request.getParameter("password");
3 Statement st = conn.createStatement();
4 String query = "SELECT * FROM users WHERE username='" +
5                 username + "' AND password='" + password + "'";
6 ResultSet rs = st.executeQuery(query);
```

Figure 1. Vulnerable code in a Java login servlet susceptible to SQL injection

The example is taken from a login servlet that gets two user submitted parameters, *username* and *password*, and uses them directly in a dynamic SQL query. The injection is accomplished if the WHERE clause can be evaluated true without providing a matching login credential. This can be done by adding *' OR '1'='1* in the *password* parameter because the OR condition will always evaluate true.

A way to detect information flow based security flaws is by performing *static taint analysis*. The idea is that variables that directly or indirectly can be modified by the user is identified as tainted. If a tainted variable is used to execute critical commands a potential security flaw is detected. In the example in Figure 1 the method *request.getParameter(...)* gets user input and the variable in which the data is stored is identified as a *source* in taint analysis. On the other side of the information flow is the method *st.executeQuery(query)*, which is an endpoint executing a critical command, identified as a *sink* in taint analysis.

In developing applications in general a popular approach to work by is to implement some kind of agile software development methodology. The main agile practice that this article is highlighting is *Iterative and Incremental development*. Being iterative means that the current state of the developed functionality is improved, adding quality to the code for a better product. Incremental development refers to breaking up the work into smaller pieces. The pieces are scheduled to be developed, usually in timeboxed cycles, and integrated in the software as they are finished. This could also be done as a response to new or changing requirements.

When using agile software development methodologies, a principle from the Agile Manifesto is worth mentioning, *deliver working software frequently*. This principle encourages to develop functionality in small time frames so that the customer frequently is presented the latest product increment. For the developers it is tempting to maximize the deliverance of functional requirements if the customer has not communicated an emphasis on non-functional requirements such as software security.

B. Problem Description

The main objective of this article is to study how to integrate static taint analysis in an iterative and incremental development process to detect information flow based security vulnerabilities in Java EE web applications. This integration was proposed by the authors in an earlier work [2], but in this paper, we apply the proposed principle to an actual development process.

A typical Java EE web application featuring several different components will be developed in order to attain practical experience using the technology. This application will be

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

in use by a company in an industrial environment possibly accessed through the internet. Potential security flaws may expose confidential data through information flow security vulnerabilities.

Because of the principle of frequent delivery in agile software development methodologies, integrating security analysis should be as cheap as possible in regards to the process of analysis. The following aspects will be considered.

- Resources needed in form of preparing the application for analysis parameters before execution of the taint analysis
- Typically, how much time is needed to run the taint analysis for a considerable large web application
- Resources needed in order to interpret and respond to the taint analysis output

C. Article Outline

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, other works related to this and state of the art. In Section III, the methodology used in this study is presented. Then, an actual implementation of the proposed method is demonstrated in Section IV. 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. Theoretical Background

A. Software Development

When developing software, a common approach is to establish a *Software Development Life Cycle (SDLC)*. The SDLC's function is to cover all processes associated with the software developed. Different types of SDLC models exists. However, whether it being Waterfall, Agile or some other model the processes in the SDLC can be identified in five phases. The phases are named *Requirements*, *Design*, *Development*, *Test* and *Deployment* [3].

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.

Requirements phase addresses the gathering and analysis of requirements regarding the environment in which the software is operating in. 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.

Design phase is where the functional requirements of the software developed is planned based on the mapping of requirements in the first phase. This phase also includes architectural choices that determines the technologies used in the development of the software.

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 usual approach is to develop the functional requirements in small programs called units. These units is then tested for their functionality, called *Unit Testing*.

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. And when fixed, the system would need to go through the test phase once again.

Deployment phase is the final phase that exists 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 [3].

B. Software Security

OWASP analyse 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 resides numerous exploitable implementations such as queries for SQL, Lightweight Directory Access Protocol (LDAP), Xpath or NoSQL and command injection in form of operatings 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 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 *insecure direct object references* in the OWASP top 10 [1] [4].

HTTP response splitting is a technique that involves splitting the HTTP response enabling an attacker to gain control over the second HTTP response. The HTTP response could be split if the application includes malicious data in the HTTP response header. Simply supplying a line break (CR and LF) in the malicious data splits the response. Implications includes

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Figure 2. The Software Development Life Cycle [3].

web cache poisoning, cross-user defacements, page hijacking and cross-site scripting [4].

C. SQL Injection

SQL injection can be further broken down into different types of injection techniques. *Tautology* is a technique to bypass authentication and access data through the *WHERE* clause by making the query always evaluate to true. The SQL injection example shown in Figure 1 is an example using that technique. The following SQL query is a general SQL injection example of the tautology technique. [5].

*SELECT * FROM <tablename>WHERE userId = <id>and password = <wrongPassword>OR 1=1;*

A technique used in order to force the application to display an error message sent from the database is called *logically incorrect queries*. The idea is to make the SQL query fail in order to acquire information about the database structure such as table and column names. If the application does not withheld such error messages from the users an attacker could learn enough to forge an effectively targeted SQL injection to access or modify the desired data. In the following example SQL query an additional *query delineation (')* is added after the username parameter rendering the SQL query incorrect. In this specific case the error message would reveal the name of the password parameter [5].

*SELECT * FROM <tablename>WHERE username = <anyUsername>' and password = <anyPassword>;*

The next technique, named *union queries*, uses the *UNION* clause in order to acquire information from other tables in the database. In the following example an attacker needs a valid user and password pair. The attacker adds the extended query in the password field after the valid password. This example fetches the current user's credit card number [5].

*SELECT * FROM <tablename>WHERE userId = <id>and password = <rightPassword>UNION SELECT creditCardNumber FROM CreditCardTable;*

Piggy-backed queries is a technique to expand the number of SQL queries the DBMS would execute by using the *query delimiter (;)*. The first query is the former query, which will be executed normally and the following queries that are added by the attacker are also executed. Since an attacker could construct any SQL query, the possibilities for exploitation are extensive. Some outcomes could be adding, modifying or deleting data, performing denial of service and executing remote commands. In the following example an additional query is constructed deleting a table [5].

*SELECT * FROM <tablename>WHERE userId = <id>and password = <rightPassword>; DROP TABLE <tablename>;*

The preceding SQL query examples are the simplest SQL injection techniques. A couple of other techniques are also worth mentioning that are slightly more advanced. *Blind injection* could be used to acquire data if the application is hiding database error messages from the attacker. The concept is to query the database with queries evaluating true or false in order to slowly accumulate information by elimination. The prerequisite for this to work is to find a way to tell whether the query evaluates true or false. This could be done in e.g., a login context [5].

In case a way to tell if the evaluation is true or false is not found, the next technique could be applied, namely *timing attacks*. With the help of an if-then statement and the *WAITFOR* clause a delay could be set depending on how the query evaluates. E.g., a database delay for 5 seconds could be set if the query evaluates true and otherwise have no database delay. By observing the response a conclusion could be made in whether the query evaluated true or false [5].

D. Cross-Site Scripting

Cross-site scripting is a vulnerability that enables the attacker to get a user visiting an infected website to run malicious scripts. Some outcomes for the attacker is hijacking the user's session, redirecting to other websites and modifying the compromised website's presentation of its content. Three types of cross-site scripting attacks exists. *Non-persistent attacks* is the most common type and is an attack that is not stored persistently, but reflected to the victim immediately. An approach is that the attacker sends a URL to a vulnerable, but seemingly trustworthy, web page. The link contains a malicious script that will be executed if the victim clicks it. Consider the following example URL exploiting a web page search field susceptible to cross-site scripting because a lack of validating both user input and output. The user input is the search string and the output is what is outputted in the search results web page [5].

http://vulnerable.site/search.php?query=<script>alert(0)</script>

In this example the script is only triggering an alert box. This script could be crafted in order to steal the victim's cookies, session or other accessible information. The second variation of cross-site scripting is called *persistent attack*. Instead of crafting a malicious URL this technique goes hand in hand with injection in that the malicious script are stored persistently, e.g., in a database. The attacker first injects the malicious script in the vulnerable web site and the victim visits the web site serving the script at a later point in time. An example could be a message board where users posts messages accessible by other users [5].

The third cross-site scripting technique is called *DOM based cross-site scripting attack*. This approach is different in comparison to the other techniques in that it is a client side issue. The idea is to manipulate the Document Object Model (DOM) by injecting malicious data into the website, e.g., inserting a fake login form tricking the victim into submitting sensitive information. Web sites are vulnerable to this type of attacks because input are not validated and escaped properly.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A web site could have good validation routines server side, but since this type of attack opens for purely client side manipulation validation of input data client side is crucial [5].

E. Mapping Threats

In order to eliminate security flaws when developing a web application a crucial point is that the software developers need to have an idea of how the web application is vulnerable. An option is to use a *threat analysis*, which fits in the design phase of the SDLC. The analysis typically consists of three steps. The first step is to determine and categorize the system's possible threats. Then each threat are ranked by the expected security risk. Finally, a mitigation plan regarding the ranked list is laid out [3].

Another concept that can be used in threat analysis to identify threats is mapping the application's *attack surface*. An attack surface is defined as all possible entry points an attacker can use to attack the application. In a web application all web pages the attacker can access contributes to the attack surface. For the information flow based vulnerabilities included in the threat analysis a mitigation plan could contain specific design choices in order to counter these threats. However, additional initiatives should be included in the testing phase to make sure any developer mistakes are caught [3].

An approach in order to catch developer mistakes is to initiate a manual code review by security experts. This strategy, although usually highly effective, is both expensive and time consuming. Automatic detection of vulnerabilities in some form is the preferred way to go.

F. Methods for Detecting Vulnerabilities

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 a SQL query. Based on this analysis, test cases are generated in form of unit tests with attack input. Lastly, the test cases are executed and a test result summary showing vulnerable code locations are provided [6].

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 are in fact moved from the application layer to the database layer. Any attempt to execute SQL injection cannot affect the data of different users. [7].

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 [8].

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 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 [9].

The second policy, *DOM sandboxing*, is made to enable the use of unknown scripts. This could be a necessary evil for a web site for e.g., requiring 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 an <*div*> or <*span*> element enables sandbox mode in that element disallowing running scripts [9].

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 whether it being either SQL injection or cross-site scripting. Since both types of vulnerabilities belong to the same category of vulnerabilities, information flow vulnerabilities, a mutual approach is desirable to explore. Such approach should also be able to detect all forms of information flow vulnerabilities.

FindBugs 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. FindBugs states the following:

"Because its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50% [10]."

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 developer's time. Therefore, finding a method with a much higher accuracy is preferable. The approach this article are looking into in order to detect information flow vulnerabilities is an approach called *taint analysis*.

G. 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 needs to be identified. The external inputs could be data from any source outside the application that is not trusted. In other words where there is a crossing in the application's 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.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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

Figure 3. A tainted source variable containing an id to fetch data from a HashMap indirectly induces taint on an object [11]

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 that a tainted variable that contains an id is used to determine what data is assigned to a new variable, see Figure 3 [11].

Tainted variables in itself are not harmful for any applications. It is 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 a way that it will not contain any data that could exploit the critical command in which it will be used. An example is that when querying a database with a tainted string it could open for SQL injection if the string contains characters that either changes the intended query or splits 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. Consider the SQL injection example in Figure 1 the method *request.getParameter(...)* reads input from the user. This input is stored in a string making it tainted. On the other side of the information flow is the method *st.executeQuery(query)*, which is a sink. When the tainted source string are used in the sink as part of the SQL query without sanitization an information flow vulnerability is evident.

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 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 analyse 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.

The concept of taint analysis has been around for several decades. The scripting programming language Perl introduced *taint mode* with Perl 3 in 1989. Taint mode is implemented as a native feature in Perl's interpreter and is enabled if the Perl script runs with the *-T* switch. When taint mode is enabled all strings that originates from outside the program are marked as tainted. If a critical operation is executed with a tainted string the program fails with an error. Examples of sinks are methods to write to files, executing shell commands and sending information over the network.

In order to enable the use of tainted strings in sinks, Perl taint mode policy is that the string needs to be untainted. This process consists of sanitizing the string by using regular expressions. Consider the use of regular expression in order to e.g., remove a trailing character. In this case the developer needs to be aware that doing this removes the taint from the string. The lack of further sanitizing of the tainted string renders it with an improper sanitization for safely being used in sinks.

The Perl taint mode implementation is a dynamic approach since the analysis tracks tainted strings in the program's runtime environment. Dynamic taint analysis has some variations in areas of use and Perl taint mode resides within the category *unknown vulnerability detection*. This is, as shown with the Perl taint mode example, simply detecting misuses of user input during execution with the goal being preventing code injection attacks [12].

Further, dynamic taint analysis can also 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 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 [12].

Taking this analysis one step further enables malicious software detection of e.g., keyloggers, packet sniffers and stealth backdoors. The concept being marking input from keyboard, network interface and hard disk as tainted and then tracking 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 are enabled. E.g., in the case of detecting keyloggers, the profile includes which modules that normally would access the keyboard input on a per application basis. When a keylogger is trying to access a specific profiled application this could be detected [13].

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 truly not 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 absent of an actual error [12].

Dynamic taint analysis has, as shown in previous paragraphs, several types of applications. However, static taint

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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. Three of them, which are implementations for Java, are elaborated on in the following sections.

H. Taint Analysis for Java

The first implementation, *Taint Analysis for Java*, 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. The paper describes an implementation of specific algorithms, but the analysis design is flexible in that using any set of desired algorithms are feasible [14].

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 traditionally *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. [15].

Thin slicing works well with taint analysis because the statements most relevant to a tainted flow is 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 the 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 [14].

The way this implementation defines sources and sinks is through *security rules*. Security rules exist on 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 [14].

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 specially treated. These classes contains *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 [14].

Refer to Section II-K1 for more information on the article describing Taint Analysis for Java.

I. Tainted Object Propagation Analysis

The second 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, refer to Section II-H. 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 [4].

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^{14} contexts. This allows the analysis implementation to scale to applications consisting of almost 1000 classes [4].

In order to detect vulnerabilities, specific vulnerability patterns needs 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) [4].

Tainted Object Propagation Analysis does not implement any handling of web application frameworks. Refer to Section II-K2 for more information on the article describing Tainted Object Propagation Analysis.

J. Type-based Taint Analysis

The 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 that 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 with each computed value in the type system and the flow of these values are then examined. This concept is called *subtyping* [11].

The subtyping hierarchy is defined as *safe* $<:$ *tainted*. This means that a flow from tainted sources to safe sinks are disallowed. The other way around, assigning a safe variable to a tainted variable, is allowed. For an example of annotation, refer to Figure 1 where the source *request.getParameter(...)* would be annotated as tainted and the sink *st.executeQuery(query)* would be annotated as safe [11].

A third type qualifier, *poly*, is included in order to correctly propagate tainted and safe variables through object

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 inherits the manipulated inbound parameter's type qualifier (tainted of safe). As a result the subtyping hierarchy becomes *safe* $<:$ *poly* $<:$ *tainted* [11].

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 regular Java API is supported, namely by annotating the relevant fields and methods. An example 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 tainted would include the framework in the analysis [11].

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. Refer to Section II-K3 for more information on the article describing Type-based Taint Analysis [11].

K. Related Work

In choosing which articles to be included as related work, the emphasis is on articles describing practical taint analysis implementations backed by analysis results. Theoretical implementations can be a good starting points in expanding a research topic. But since the main goal of this article is to study how taint analysis can be integrated in a development process an actual working taint analysis implementation is preferable.

1) TAJ: Effective Taint Analysis of Web Applications : This paper describes the design and implementation of a static Taint Analysis for Java (TAJ). The use of pointer analysis and the construction of a call graph is the first step in the analysis. Further, a hybrid thin slicing algorithm is used to create a Hybrid System Dependence Graph (HSDG). Finally, computation of reachability in the HSDG is conducted in order to find tainted flows. Scalability is built in enabling analysis of applications of any size with the help of a set of techniques designed to produce useful results given limited time and space. Techniques included are priority-driven call graph construction and using bounds on other parts of the analysis, e.g., to limit the size of a slice in the hybrid thin slicing algorithm [14].

TAJ was designed to support a commercial product, IBM Rational AppScan Developer Edition (AppScan DE), and has therefore undergone extensive evaluation. 22 different applications that mostly make use of web frameworks are analysed using 5 different variations of the thin slicing algorithm. This was done in order to identify an efficient compromise on performance and the number of false positives present because the analysis introduces a high percentage of false positives. However, few false negatives are reported by the analysis [14].

2) Finding Security Vulnerabilities in Java Applications with Static Analysis : This paper proposes a static taint analysis implementation based on a context-sensitive pointer analysis. Based on the pointer analysis a call graph is generated. The paper describes the class of information flow vulnerabilities as the tainted object propagation problem. Users need to provide a specification of which methods that can lead to a vulnerability in the form of different types of descriptors. The specifications are automatically translated into static analysers. Results of the analysis are presented as a plugin for Eclipse IDE enabling examination of each vulnerability found [4].

It is reported that this analysis scales to programs of almost 1000 classes. Further, the analysis is done at the bytecode-level meaning that the approach can be applied to other forms of bytecode, e.g., enabling the analysis of C# code. There is no information on how this analysis can include other web application frameworks other than the standard Java EE implementation [4].

The analysis was run on nine popular open-source applications resulting in 29 detected vulnerabilities. Two of the vulnerabilities resided in widely-used Java libraries. Further, the analysis yielded 12 false positives, however, all false positives came from one of the nine applications. The authors concluded that their approach yields very few false positives [4].

3) Type-based Taint Analysis for Java Web Applications : This paper presents a type-based taint analysis approach. SFlow, a context-sensitive type system for secure information flow is implemented in a checking framework that the authors has built in previous work. This framework infers and checks object ownership and reference immutability. Users need to annotate sources and sinks, and the analysis runs without further input from the user reporting either a concrete typing or type errors indicating information flow vulnerabilities [11].

The taint analysis approach handles reflection, libraries and frameworks effectively. Handling reflection is possible because SFlow does not require abstraction of heap objects, as the flow is tracked through subtyping. Both libraries and frameworks are also handled through subtyping together with the fact that the analysis is modular. This means that it can analyse any given set of classes. If the set contains an unknown callee, e.g., a library method with unknown source code, both source and sink information flow are correctly tracked through subtyping [11].

Evaluations that are performed on 13 relatively large Java web applications have shown both precision and scalability. It has zero false positives for most of the applications and about 15% false positives on average. An indirect comparison with TAJ, [14], and F4F, [16], was done in that both implementations are included in the commercial tool AppScan Source. In addition, another commercial tool was also included in the comparison, Fortify SCA [11].

AppScan Source and Fortify SCA detect respectively 50% and 61% of all vulnerabilities, while SFlow detects 100%. The precision is 74% for AppScan Source, 81% for Fortify SCA and 76% for the SFlow implementation. Precision P is

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

defined in the following way where T indicates the number of true positives meaning the correct detections, and F being the number of false positives [11].

$$P = \frac{T}{T + F}$$

4) F4F: Taint Analysis of Framework-based Web Applications : This paper describes F4F (Framework For Frameworks), which is a framework for conveniently adding support to framework-based web applications in taint analysis. Since framework implementations extensively use reflection, conducting static taint analysis are often unable to detect vulnerabilities correctly. F4F presents a way to generate a specification of a program's framework-specific behaviours and integrate this specification into the taint analysis engine without changes to the underlying analysis engine. [16].

The approach F4F uses is to utilize the Web Application Framework Language (WAFL) in order to generate specifications. Futher, a helper tool, WAFL2Java, translates the WAFL specifications to Java code for use with the taint analysis implementation. A higher-level API for generating WAFL specifications is also implemented easing the process of writing WAFL generators. WAFL specification support is added to the taint analysis implementation ACTARUS, which is an improved version of TAJ, refer to Section II-H and II-K1. TAJ includes a built-in framework support that is not included in ACTARUS. However, the combination of ACTARUS and F4F discovers more framework-related issues than TAJ [16].

F4F is evaluated by analysing nine subject programs. The set of programs exercises four supported frameworks. Eight use Struts, three use Spring, five use Tiles and six use EL. F4F detected 525 new vulnerabilities compared to ACTARUS taint analysis without F4F support. The number of more vulnerabilities detected per program ranged from 1.1X-14.9X with a harmonic mean of 2.10X. A manual inspection of the new vulnerabilities detected revealed that many were exploitable or reflected bad security practice [16].

5) Related Work Conclusion: Three practical implementations of taint analysis are included as related work. The analysis methods for those implementations are described in detail in Section II-H, II-I and II-J. Other variations of taint analysis implementations exists, however, limiting to these three implementations covers the most important methodologies present in the domain of taint analysis implementations. Also, these implementations are crafted specifically for Java in order to fit with analysis in the context of Java EE web application development. Taint analysis implementations exists for numerous programming languages, especially for the C/C++ programming language.

In addition to the taint analysis implementation articles an article describing the use of taint analysis in framework-based web applications is presented. The Java EE web application implementation is in itself a framework and it can also be extended by third-party framework implementations. Frameworks introduce a layer of added complexity and it appears to be a challenge to properly cover frameworks in static taint analysis implementations. With this in mind and an overview of different taint analysis implementations the course is set in deciding the methodology.

III. METHODOLOGY

Taking a brief look at the core of the problem description, refer to Section I-B, it is stating that this article will study how to integrate static taint analysis in Java EE web applications. Refer to Section II-F briefly stating some proposed methods for detecting information flow vulnerabilities, static taint analysis is explored in this article. Both because this type of analysis embraces the detection of the whole domain of information flow vulnerabilities. And that it may have significantly fewer false warnings in comparison 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 in regards 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.

For this type of article, why is such a development of a prototype application necessary? One could simply argue that using an open source Java EE web application as the artefact for performing taint analysis is equally sufficient. However a clear advantage is that when developing a new application the developer gains an exceptional understanding of all the inner workings of the application. E.g., knowing exactly which technologies are used, how the application should function and also be aware of all system critical commands implemented in the application.

The goal of the last part in the case study is to architect a solution to the taint analysis implementation. Many aspects regarding this implementation 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 would be drawn.

Section II-G describes different approaches implementing static taint analysis and thus is the basis in choosing the analysis method. The first two implementations described are not freely available for use. However if one of those approaches had been in any way superior to the third alternative, Type-based Taint Analysis, an effort to acquire the implementation might have been worth it. The choice of analysis method is as implied the Type-based Taint Analysis. This choice is convenient in that the analysis platform is available as an open source project.

Type-based Taint Analysis also looks promising due to how web application frameworks are handled. Analysing frameworks are especially relevant in Java EE web applications, e.g., in form of the Java Server Faces (JSF) framework managing the application's front-end. Based on how the article are describing this analysis method it would seem that the implementation is feasible as an integrated step in in a Java EE web application development context, refer to Section II-K3.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

IV. Prototype Application Development

In this chapter a description of the prototype application is given. Additionally, an overview of the development process for the prototype application is covered. Security aspects in form of the prototype application's attack surface is also discussed.

A. Existing System

The prototype application is an application that is going to replace a standalone SMS alarm system used at Findus' food plant in Tønsberg, Norway. Currently, this SMS alarm system is used to notify personnel of irregularities affecting the production environment.

Two main applications of alarms are set up. The first application covers fire and gas alarms. Fire alarms are of course mandatory in any industrial building. However, gas alarms are present because the cooling systems are using hazardous coolants. Both alarms are prior notification alarms. This means that when detectors are sensing increased levels of either smoke/heat or gaseous particles respectively, alarms are delivered to the desired people. This could in some situations buy some time for investigating and reacting before the actual alarm is activated triggering a call for the fire department.

The other application of the SMS alarm system is to manually notify personnel, e.g., the supervisor or technical assistance, of incidents regarding the process machines. These types of alarms are mainly triggered by mechanical push buttons located near critical places such as vegetable cutters, conveyor belts and transport pumps.

B. Limitations

The standalone system is limited in how alarms are connected. It does not support any kind of communication buses. This means that every alarm needs to be electrically connected resulting in a strict limitation to the number of alarms possible. This has greatly prevented expansion of the system. The system supports both digital and analogue alarm inputs limited to 20 digital and 8 analogue. Digital alarms are typically connected to either a manual push button or a relay output from e.g., a fire alarm system.

Analogue alarms could be anything providing a measurement such as a temperature sensor or a level indicator. Contrary to a digital alarm triggering an alarm based on a binary signal, an analogue alarm needs a set point indicating when an alarm should trigger. Additionally, information whether the alarm area of the analogue signal should be triggered above or below the set point is also required. All 20 digital alarm inputs are used, however, at the present time no analogue alarms are in use.

Apart from the limited number of digital inputs the main limitation is the connectivity process. Electrically connecting alarm signals are expensive. Both because it is a time consuming task to physically connect an alarm signal to the alarm system and expensive in regards to the cost of cables and the occupation of a relay output.

Another limitation greatly preventing expansion is that the alarm system has its phone book filled up. The alarm system supports a maximum of 8 phone numbers. This means that only 8 individuals has the possibility to receive alarm messages. Further, no way of customizing the format of the alarm message is possible. The alarm text simply shows the alarm input number and a customizable alarm name consisting of maximum 12 characters. Finally, it is worth mentioning that the configuration of the alarm system is a cumbersome process. Two possible approaches are supported. Either through sending SMS command messages or connecting a computer to the system's serial configuration interface.

C. The New Alarm System Architecture

This section covers the initial planning phase for developing the new SMS alarm system ideally countering the limitations of the existing system by choosing an appropriate architecture.

D. Available Resources

In order to develop the SMS alarm system an initial assessment of Findus' existing resources and infrastructure are mapped. This kind of mapping is done to be able to design a system with the ability to utilize the available resources reducing cost and overall complexity. Refer to Section IV-B stating that the main limitation of the original alarm system is the alarm input connectivity.

A look at how information from process machines is accessed reveals that the use of a Supervisory Control and Data Acquisition (SCADA) system is in place. This system monitors and controls the process machines over Ethernet network connectivity. This is possible because the process machines are controlled by Siemens S7 Programmable Logic Controllers (PLCs) equipped with Hilscher netLINK NL 50-MPI adapters. This adapter enables Ethernet connection to the process machines by acting as an Multi-Point Interface (MPI) node on the PLC's MPI network.

Utilizing the same process machine Ethernet communication as the SCADA system would eliminate both the limitations regarding the maximum number of alarm inputs and the expensive connectivity process. When it comes to running the new SMS alarm system there is currently free server capacity within the technical network, where also the SCADA system resides.

E. Sending SMS Alarm Messages

The utilization of existing process machine Ethernet communication and server capacity goes a long way. However, the process of sending the SMS alarm messages needs a solution. Two options are possible. Either using an external SMS messaging service provider or sending SMS messages with the help of a standalone GSM modem. An SMS messaging service provider would be the easy solution simply requiring a subscription and an implementation of the service's SMS message Application Programming Interface (API). The GSM modem approach requires a dedicated SIM card with an active subscription and a complete implementation of the GSM modem communication in the new alarm system application.

Even though less work is needed with an SMS messaging service provider it comes with some disadvantages. It would

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

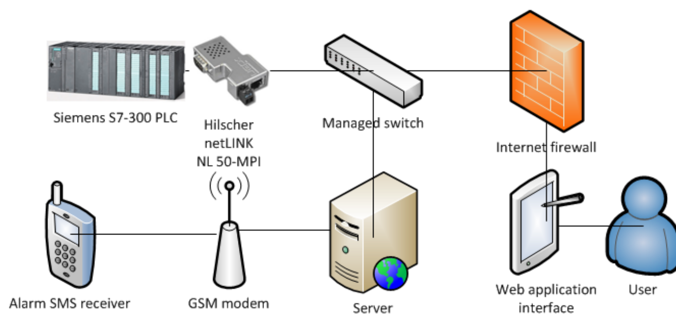

Figure 4. Architectural sketch of the components in the new SMS alarm system

require an internet connection at all times and would also introduce an external dependency. Some risks regarding this dependency is service downtime and delays reducing the quality of the alarm SMS messages. An option in eliminating these risks is to implement a failover routine to a second SMS messaging service provider. This will lead to another disadvantage: increasing the system's complexity. And since GSM modems are available at an inexpensive price, the GSM modem approach is preferable.

F. Architectural Sketch

The initial planning phase leads to an overview of the different components the new SMS alarm system should consist of. See Figure 4 for an architectural sketch of the component composition. Two more concepts are added in addition to the components established in the preceding sections. The first concept is a user and a web application interface that represents that the configuration of the SMS alarm system would be done with a web application implementation. The second concept is the alarm SMS receiver, which typically is a mobile phone exactly like it is done in the original alarm system.

G. Development Technology Choices

Technologies chosen need to be able to meet the architectural design requirements established for the application. The framework chosen for developing the prototype application is Java EE with the Java Server Faces (JSF) user interface framework. By choosing a web application framework it is easy to implement web pages for configuring the prototype application. The goal for this implementation is to eliminate the limitation regarding the cumbersome process of configuring the original alarm system, refer to Section IV-B.

The IDE that will be used is Eclipse and the Java project for the prototype application is managed by Maven build system. An advantage with using Maven build system is that it contains a definition file for defining which libraries that should be included in the Java project. Building the application will automatically download these dependencies from the Maven repository. By using such a system it is straightforward to deploy the application elsewhere, e.g., to a production server.

H. Development of the Prototype Application

Although a fully-fledged SDLC methodology was not followed given the in this project, several concepts were integrated in the SDLC in order to ensure deliverance of an acceptable end product. Concepts derived from the agile manifesto core value *customer collaboration over contract negotiation* were embraced. Initially, this means that the design choices and functionality of the prototype application were discussed and determined through periodic communication with the industial partner.

Further, enabling development of the prototype application iteratively and incrementally was done by embracing 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. When developing in this way a common approach is to have a test server, called a continuous integration server, for deployment. The test server is a temporary server that is as similar as possible to the production server. For the prototype application this means that access to a PLC and connection to a GSM modem was provided.

The continuous integration tool chosen is Jenkins. This is a tool for use on a continuous integration server and it was installed on the test server. It was configured to automatically deploy the prototype application to Apache Tomcat, which is the Java servlet container used. This works by pushing code to a version control system, such as Git. When Jenkins detects a change in the Git repository, the source code is pulled, built and deployed on the test server.

Additionally, the Jenkins plugin SonarQube was set up to automatically run on every build. This plugin includes various code checking tools in order to improve the code quality by suggesting changes. FindBugs, briefly mentioned in Section II-F, is one of the tools included in SonarQube.

In order to establish the course of how the development of the prototype application was conducted user stories was the main tool used. This is a tool derived form the agile SDLC methodology and is a brief description of a requirement stating the desired feature. In other words splitting up functionality into smaller manageable pieces fitting well in a continuous delivery context. The functionality stated in the user stories originated from collaboration with the customer.

Finally, a concept also used is sprints. Each sprint is a defined period of time in order to complete a set of user stories. The time frame of each sprint was defined to two weeks. The application was developed in six sprints and had a total of ten user stories defined. A brief description of the main modules are provided in the following sections.

The prototype application can roughly be divided into the following main modules.

- Communication with PLCs
- Sending triggered alarm SMS messages
- GUI for configuration of parameters

I. Communication with PLCs

The application communicates with the configured PLCs through Ethernet in the same way the SCADA system does. All PLC parameters that are configured for acquiring are fetched and stored in a database residing on the same server as the

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

prototype application is running. The parameter update interval is customizable and it is set to one second as the default setting. It is worth noting that the PLC communication is conducted unencrypted on a dedicated VLAN on the technical network. Encrypted communication is not supported by the MPI to Ethernet adapters currently in use.

J. Sending Triggered Alarm SMS Messages

This module is twofold in that first a determination if an alarm is triggered are carried out and secondly if an alarm triggers an alarm SMS message are sent. There are different settings for each configured alarm defining the respective alarm rule. Based on a comparison of the last and current parameter value in light of the current alarm rule a decision about whether or not the alarm triggered is made. On triggering of an alarm, an SMS message is sent to the assigned phone numbers for that alarm in particular. This is done using the GSM modem connected serially to the server.

K. Interface for Configuration of Parameters

The application's parameters are customizable using web pages as the interface and the database as parameter storage. The main functionality in the interface is the possibility to define which PLC variables to retrieve, phone numbers to be used, definition of alarm triggers based on PLC variables available and mapping of the defined phone numbers to the specified alarm triggers. Further, application specific parameters is also included, such as alarm trigger check interval, PLC variable update interval and default alarm message.

The GUI also includes monitoring of the PLC variables fetched including a time stamp, logging of sent alarm messages and the health status of the GSM modem in form of the GSM signal quality. An utility for sending custom SMS messages is also added to the interface. This is done in order to have a convenient way of notifying users of special events, e.g., system downtime.

Lastly, it is worth mentioning that the GUI is implemented with a basic built in login routine provided by the application server, Apache Tomcat. All interactions with the configuration web pages is also strictly enforced to always use encrypted transport protocols, namely Secure Socets Layer (SSL).

L. Prototype Application's Attack Surface

Analysing potential areas in the prototype application that has a possibility of introducing vulnerabilities is advantageous in that an awareness in those areas are raised. Preferably this awareness could lead to implementing measures countering potential vulnerabilities. A method in doing this is to conduct a mapping of the attack surface, refer to Section II-B.

In order to map the prototype application's attack surface knowledge of what features the application has implemented is required. For the prototype application a look at all the system critical operations is the first step. These operations could be seen as the attack target in the application and are covered in the following list.

- PLC communication to acquire process machine data
- GSM modem for sending alarm SMS messages
- Database connection for storing process machine data, sent SMS messages and all configuration parameters on the configuration web pages

The listed attack targets can ultimately be exploited in different ways. Both the PLC communication and the database connection can pose vulnerable to information leakage and data manipulation, while the GSM modem could be hijacked. The next step is to analyse the different entry points that contributes to the total attack surface. It is the entry points that enables a potential attacker to reach the critical operations. The following list contains possible entry points.

- All configuration web pages
- The server hosting the prototype application and the database
- VLAN where PLC communication is conducted

In the following subsections aspects around each mapped entry point are discussed. This includes thoughts about how the application is vulnerable and suggested countermeasures.

M. Configuration Web Pages

The prototype application is developed for the possibility of being accessed through the internet. Being accessible through the internet contributes to this entry point being highly exposed. With this consideration in mind security measures such as a login system and the use of encrypted transport protocols are implemented in the application.

The login system used is basic authentication, a login implementation included in Apache Tomcat. For this prototype application login credentials are simply statically added to the configuration file *tomcat-users.xml*. Accessing any of the configuration web pages renders a pop-up dialogue requiring a valid user name and password combination.

Possible weaknesses that can lead to a vulnerable system for this login implementation is brute force attacks, eavesdropping the login credentials and social engineering. In order to counter brute force attacks Apache Tomcat has an option to restrict access after a number of unsuccessful login attempts. The default settings when implementing this feature is that the user is locked for 300 seconds after 5 unsuccessful login attempts. As for eavesdropping the login credentials this can be possible if a third party has access to the communication data. Countering this is done by restricting the communication for the prototype application to always use an encrypted transport protocol.

Having taken these technical considerations a last threat cannot easily be avoided, namely social engineering. The number of possible entry points increase proportionally with the amount of users having access to the system. That being said, as this topic is slightly out of scope of this article, apart from being aware that this adds to the attack surface no consideration on this point is taken.

N. Application and Database Server

The prototype application and its database server is hosted locally on a server owned and maintained by the company itself. The entry points regarding this server is by gaining

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

access to the server locally and through one of the VLANs on the local network that it uses. With self-maintained servers patching routines for fixing new vulnerabilities is important in order to make these entry points the least possible accessible.

A special threat regarding the prototype application that is worth mentioning is the GSM modem. With access to the server comes direct access to the GSM modem enabling an attacker to abuse any features possible by the GSM modem.

O. PLC Communication

The PLC communication could be accessed either through the server or the dedicated PLC communication Virtual Local Area Network (VLAN). The threat in conjunction with these entry points is the acquisition of information about the company's processes through the PLC communication. Further, if the attacker has the possibility to transmit custom network packets on the PLC communication VLAN the attacker would also have the ability to write data to the PLCs. This means essentially to have full control of the process machines in the company.

Since the communication with the PLCs are currently conducted unencrypted and with no means of authentication countermeasures for this entry point are limited to securing the PLC communication VLAN in the best possible way.

V. Analysis and Assessment

A. Role of the Prototype Application

The prototype application would ideally be developed in iterations with an integrated taint analysis implementation as a part of the static analysis step. However, the application was fully developed before the taint analysis implementation was set up. This means that the taint analysis is carried out after all development iterations are finished.

As a result the experiences that would have been acquired conducting taint analysis in the developing phase are absent. Since the prototype application is limited in size with a moderate number of iterations conducting taint analysis at the end of the development are considered adequate in order to draw a conclusion. 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 progress. Thus saving developer resources.

B. Type-based Taint Analysis Implementation

Refer to Section III stating that the type-based taint analysis approach is used in this article. This implementation is called SFlow, refer to Section II-J for a brief overview of the concepts. SFlow is made open source using the Apache License (version 2.0), available at *github.com/proganalysis/type-inference*. It is built as a compiler plugin to *The Checker Framework*. This framework enhances Java's type system in order to detect a broader domain of errors at compile time. In addition to command-line usage The Checker Framework is also available as a plugin supporting various build systems and IDEs. Two popular IDEs worth mentioning in this regard are IntelliJ and Eclipse [11].

```
1 package javax.servlet;
2 import checkers.inference.sflow.quals.*;
3
4 public interface ServletRequest {
5     /*@Tainted*/ String getParameter(String arg0);
6     ...
7 }
```

Figure 5. Annotation example identifying a method for getting a Java servlet parameter as source

In a typical development environment the use of a continuous integration tool is common practise. This tool includes automated building of the application and running any desired plugins, e.g., static source code analyses. An example for such a tool for Java web applications is Jenkins. Integrating taint analysis in the continuous integration tool and/or the developer IDE is crucial for successfully conducting taint analysis without needless overhead. In this regard SFlow looks promising.

However, in order to enable analysis with SFlow an integration with build systems and IDEs is not sufficient alone. SFlow also requires some preparations in identifying which fields and methods are considered sources and sinks to actually detect information flow errors. The Checker Framework is designed to use annotations in order to identify fields and methods in Java classes of interest.

C. Annotating Sources and Sinks with SFlow

Two approaches in annotating Java classes exists, a manual and an automatic approach. The manual approach is that the developer adds an annotation on each field or method of interest. This is a cumbersome and time consuming task with too much overhead for the developer. In addition such task is prone to errors. The automatic approach is to compile an annotated Java Development Kit (JDK), which is added to the Java classpath of the SFlow analysis.

This JDK includes libraries the project uses with sources and sinks annotated. This way the annotation process becomes a one time event and any involvement of the developers is avoided. However, all new libraries implemented that introduces new methods for entry points (sources) or executing system critical commands (sinks) would need to be annotated and included in the annotated JDK in order to detect errors for their implementations. At present time SFlow comes with support for some of the Java EE classes containing sources and sinks such as the *getParameter* method in the *javax.servlet.ServletRequest* class and the *executeQuery* method in the *java.sql.Statement* class, see Figure 1 as an usage example [11].

Annotating sources and sinks is done respectively with the annotations */*@Tainted*/* and */*@Safe*/*. Notice that for a standard Java compiler this additions to the code is unnoticed since they are in fact commented out. The idea is that any annotations specific to The Checker Framework would not brake a standard Java compilation. See Figure 5 and 6 for examples of how annotations are defined for sources and sinks respectively.

Additionally, the SFlow annotated JDK comes with annotations for the Apache Struts Framework and the Spring Framework. To be able to analyse the prototype application

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
1 package java.sql;
2 import checkers.inference.sflow.quals.*;
3
4 public interface Statement extends Wrapper {
5     ResultSet executeQuery(/*@Safe*/ String arg0) throws SQLException;
6     ...
7 }
```

Figure 6. Annotation example identifying a method for executing an SQL query as sink

fully, libraries and relevant Java EE classes missing from the annotated JDK needs to be included. In order to map which annotations are missing a look at the application's attack surface is a good indication, refer to Section IV-L. From this knowledge the annotated JDK needs to include the methods regarding PLC communication, GSM modem communication and the configuration web pages.

Regarding the PLC communication standard Java classes are used for sending and receiving data on a socket, namely the *write* method in the *java.io.DataOutputStream* class and the *read* method in the *java.io.BufferedInputStream* class. SFlow includes annotations on methods for file read and write, but does not annotate read and write methods on a socket. Although the potential for vulnerabilities maybe lower working with a socket, the fact that this opens for untrusted data that can be forged by an attacker cannot be fully avoided. Also depending on the application, external system critical commands could be evident when writing to a socket.

As for the GSM modem communication the Java Simple Serial Connector (jSSC) library is used to communicate serially with it. Data fetched could be malicious and would need to be annotated as a source and commands would need to be regarded as system critical and annotated as sinks. Similar to working with a socket the need for annotation is considered a task of lower priority.

The configuration web pages includes interactions with users and are the most obvious place to make sure of having proper annotations making this the highest priority to implement. Therefore, an attempt to annotate the Java Expression Language (EL) implementation were made. Expression Language enables the JSF user interface to interact with Java Beans. In other words it acts as a communication bridge between the front-end and the back-end for fetching user input and presenting data to the user.

An attempt to annotate the *setValue* method in the *BeanELResolver* class was made. This method takes three parameters in addition to the EL context parameter; *base*, *property* and *val*. Base is the Java Bean, property is the name of the variable to manipulate in the Java Bean and val is the user supplied data to be assigned to that variable. See Figure 7 for the annotation. Having annotated this method and compiled it into the annotated JDK, a crafted example that would be found as a vulnerability by the analysis was made in order to check if the annotation was working correctly. The analysis however did not report any vulnerabilities. Further research needs to be done in finding out how to properly annotate the JSF user interface interaction with Java Beans.

In order to prepare an application for type-based taint analysis, annotations needs to be included and compiled in the annotated JDK based on what technologies and libraries are

```
1  package javax.el;
2  import checkers.inference.sflow.quals.Tainted;
3
4  public class BeanELResolver extends ELResolver {
5      public void setValue(ELContext context, Object base, Object property,
6                           /*@Tainted*/ Object val) throws ... {
7          throw new RuntimeException("skeleton method");
8      }
9      ...
10 }
```

Figure 7. An attempt to annotate the *setValue* method in the *BeanELResolver* class for defining user input as source from the configuration web pages

```
1  package databeans;
2
3  import java.io.Serializable;
4  import javax.faces.bean.ManagedBean;
5  import javax.faces.bean.SessionScoped;
6  import javax.validation.constraints.Size;
7  import org.hibernate.validator.constraints.NotEmpty;
8
9  @ManagedBean
10 @SessionScoped
11 public class DatastoreBean implements Serializable {
12
13     private static final long serialVersionUID = 1L;
14
15     @NotEmpty(message = "Please write a name...")
16     @Size(max=255, message = "Please write a name of max 255 characters...")
17     private /*@checkers.inference.sflow.quals.Tainted*/ String name;
18     ...
19 }
```

Figure 8. Manually annotated variable in the Java Bean used to store which PLC data variables that should be fetched by the prototype application

implemented in the application. At the present time the SFlow annotated JDK supports a limited variety of technologies and libraries. Thus, in general requiring relatively much annotation work depending on the size of the application.

D. Analysis Results

Since a successful annotation for the JSF user interface interaction with Java Beans is yet to be done analysis results for the prototype application is non-existing. However as a simulated test the variables in the Java Beans that are user manipulatable were manually annotated. See Figure 8 for an example of a manually annotated variable in the Java Bean that is used to store which PLC data variables that should be fetched. The user provided variables are stored in the database.

The taint analysis detected no errors due to how the prototype application is managing SQL queries. Namely with SQL query parametrization. This means that an SQL query is set up with a predefined structure taking exactly the defined data types as parameters, see Figure 9. This predefined structure makes it impossible to split a query in multiple queries or add more parameters than intended in the query. Thus, making this approach a good practise countering SQL injection and the analysis rightfully did not detect this as an information flow vulnerability.

In order to detect a vulnerability with the taint analysis implementation the SQL query was temporary changed to the standard *executeQuery* method displayed in Figure 1. SFlow then reported the type error shown in Figure 10. The first code reference in the type error text refers to line 385 in *Database.java*. This is where the string variable used in the SQL query, named *sql*, is first initialised. Next, a reference to line 387 in *Database.java* is made. This refers to where the safe method is used with the tainted variable, which in this case is *statement.executeQuery(sql)*. Finally, the type error text reports which class the safe method originates from. In this case it is the *java.sql.Statement* class.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
1  public boolean addEntryInDatastore(String name, int plcId, int datablockNumber,
2                                      int startAddress, int dataType) {
3      if (connectDB()) {
4          try {
5              preparedStatement = connection.prepareStatement("INSERT INTO sms_alarm_system.variables
6                  (name, plc_id, datablockNumber, startAddress, dataType) VALUES (?, ?, ?, ?, ?)");
7              preparedStatement.setString(1, name);
8              preparedStatement.setInt(2, plcId);
9              preparedStatement.setInt(3, datablockNumber);
10             preparedStatement.setInt(4, startAddress);
11             preparedStatement.setInt(5, dataType);
12             preparedStatement.executeUpdate();
13         } catch (Exception e) {
14             LOGGER.log(Level.SEVERE, e.toString(), e);
15             close();
16             return false;
17         } finally {
18             close();
19         }
20     } else {
21         return false;
22     }
23     return true;
24 }
```

Figure 9. SQL query with parametrization used in the prototype application to store connection information of a PLC data variable that should be fetched in the prototype application

```
1  SUB-7466: Database.java:385(23543):VAR_sql{@Tainted}  <:
2  (23573:#CONSTANT#Database.java:387:(callsite)statement.executeQuery(sql)
3  {@Poly @Safe @Tainted} =m=> zLIB:java.sql.Statement:0(23572):PAR_arg0{@Safe})
```

Figure 10. SFlow type error showing that there is an information flow vulnerability in line 387 in *Database.java* in the *executeQuery* method

The type error identifies the tainted variable and the code location of it first being initialised as well as the location of the safe method using the tainted variable. This information is sufficient for a developer in order to understand where to start the work of countering the type error. For the SQL query example the obvious countermeasure is to switch to the SQL query parametrization approach. However, if SQL query parametrization is not a possibility, or for other kinds of type errors, a look at where the tainted variable first originated from may be necessary. In this case the developer would need to research the path of the tainted variable. This could take a lot of valuable time if the application is big. It would have been advantageous if the type error text also stated the code location in where the tainted variable originated.

E. Integrating Taint Analysis in the SDLC

Considering modern development practises are team based, and in fact multi-team based on big projects, it is important to include this observation in assessing if static taint analysis can efficiently integrate 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 *Type-based Taint Analysis for Java Web Applications* technical report 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, refer to Section II-A describing the different phases in the SDLC [11].

In addition to the development phase, the testing phase could include static taint analysis. However, the reason to avoid integration within the testing phase is that anything added to that phase adds unnecessary overhead. Even if overhead 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 II-F. Most of the 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 is done because it could detect a high number of vulnerabilities and also have a low number of false positives. Refer to Section II-J showing that a comparison of SFlow and two commercial security testing tools shows that SFlow detects a significantly higher number of vulnerabilities.

Refer to the preceding sections stating that a working implementation of SFlow is set up. Although the attempt to add the Java EE JSF framework to the SFlow annotated JDK was not successful, results exists by doing a manual annotation. A challenge with this implementation is to properly annotate external libraries, e.g., frameworks, in order to enable a working analysis without developer intervention. Manual annotations is 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 the annotation process needs to improve.

One approach in changing the annotation process is to take a concept from the paper *F4F: Taint Analysis of Framework-based Web Application*, refer to Section II-K4. 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 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 [16].

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 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 the *Type-based Taint Analysis for Java Web Applications* technical report states that analysing 13 relatively large application 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 CPU 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 CPU

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and 6GB RAM [11].

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 developer's IDE, 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, 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 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 needs 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.

Three main areas was identified: PLC communication, GSM modem communication and the configuration web pages. The configuration web pages was considered the highest priority and an attempt to annotate the *BeanELResolver* class was made. This class is used for communication between the JSF user interface and the Java beans. However, the annotation attempt was unsuccessful and more research is required in how to get the annotation working properly.

In order to obtain taint analysis results a Java bean in the prototype application was manually annotated. Changing the SQL query method from the safer parametrized method to the unsafe dynamic method was necessary in order to detect a type error with the taint analysis. The type error contains information about the code location of the tainted variable's initialization, the code location of the sink this variable is used in and from what class the sink method originates. It would also have been preferable if the type error contained information of the code location of the source variable. This could save valuable time for the developer investigating the type error.

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 much 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 in conducting the taint analysis are possible. Running the taint analysis manually in command line, integrating it in the developer's 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 tool's 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.

Considering the prototype application was finished without having a proper taint analysis implementation ready for testing, this proposition would need more research in order to draw a finite conclusion in how this actually will work in an iterative and incremental development process.

VII. Further Work

In order to support Java EE web applications using JSF user interface, the next step in the taint analysis implementation is to get the annotation of the *BeanELResolver* class to work. Either this is just an annotation task or it may reveal other fundamental challenges, e.g., how the taint analysis processes the Java beans variables in conjunction with the *BeanELResolver* class.

Further work also includes more research in the area of how it is best to integrate taint analysis in a development process. The proposed solution of integrating the analysis in a continuous integration tool's build system is worth exploring. An actual proof-of-concept implementation could be using Jenkins continuous integration tool with the Maven build system.

The nature of the cumbersome annotation work presently leads to the taint analysis implementation being for the en-

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

thusiast only. A course worth researching, as suggested, is to develop a framework for easing the process of annotating. A suggestion for even further work is to make the taint analysis implementation mainstream. An extension of the taint analysis implementation is possible because it is released as an open-source application. Developers could then contribute to the taint analysis implementation by providing working annotations of frameworks and libraries.

Such a project would contribute a working out-of-the box taint analysis tool that supports a large number of popular frameworks and external libraries for others to easily include in their development processes reducing the number of implemented information flow vulnerabilities.

References

[1] T. Lie and P. Ellingsen, "Integrating static taint analysis in an iterative software development life cycle," in Proceedings of SOFTENG 2017, The Third International Conference on Advances and Trends in Software Engineering. International Academy, Research and Industry Association (IARIA), 2017.

[2] 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

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

[4] 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.

[5] A. K. Baranwal, "Approaches to detect sql injection and xss in web applications," Term Survey paper-EECE 571b, University of British Columbia, 2012.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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

[14] 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.

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

[16] 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.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org