Eir - Static Vulnerability Detection in PHP Applications

Eir (Eira, Eyra) [peace, clemency]: *She was one of Menglod's attendants and was considered the healer of the Æsir. She was the goddess of medicine and the best of all physicians.*

The Norse Myths, 2010
Heilan Yvette Grimes

Kenneth Michael Jepsen
kjepse10@student.aau.dk

Morten Nørtoft
mnorto10@student.aau.dk

Mikkel-Alexander Vej
mvej10@student.aau.dk

Aalborg University
Department of Computer Science
Student report
Selma Lagerlöfs Vej 300
Phone 99 40 99 40
Fax 99 40 97 98
http://www.cs.aau.dk

Title: Eir - Static Vulnerability Detection in PHP Applications

Theme: Security in Web Applications

Project period:
DES10, 2/2/2015 - 3/6/2015

Project group:
des1011f15

Participants:
Mikkel-Alexander Vej
Morten Nørtoft
Kenneth Michael Jepsen

Supervisor:
René Rydhof Hansen

Copies: 5

Pages: 70

No. of appendix pages: 3

Finished: 3/6/2015

Synopsis:

This report presents a static vulnerability analysis tool called Eir, created for scanning PHP applications for XSS and SQLi vulnerabilities. The tool uses known theories in the field of static analysis. It is able to detect reflected as well as stored vulnerabilities. Using pattern matching to find storage locations, this prototype shows that it is possible to find stored vulnerabilities by matching pairs of incoming and outgoing data sets in a static analysis. The tool also looks into modeling of large frameworks to scan extensions such as WordPress plugins. Modeling a large amount of functionality made it possible to detect a large amount of vulnerabilities in WordPress plugins. Eir was able to detect 66 new confirmed vulnerabilities in WordPress plugins, where 17 of these were stored vulnerabilities.

SUMMARY

This report presents the tool Eir, which is a static vulnerability scanner for PHP applications. The tool was created using well-known theory from the field of static analysis and was created in C#. The tool expands on ideas of modeling frameworks in order to scan extensions for large frameworks such as WordPress. It also implements detection of stored vulnerabilities such as stored Cross-Site Scripting. Furthermore, the tool was created with quality and extensibility in mind. This is amplified by a large amount of tests and the general quality of the code base.

The major contributions of Eir is the capability of modeling advanced frameworks for use in analyzing framework extensions as well as the capability of detecting stored vulnerabilities, such as stored Cross-Site Scripting. The tool is also created to be beneficial not only within the field of security, but in the field of static analysis of PHP code in general. This is because the individual components of the tool can be reused individually for different purposes.

Eir takes PHP code and transforms it into a Control Flow Graph (CFG). This CFG is utilized in the taint analysis created in order to track taint through the code. The tool uses the theories of monotone frameworks and utilizes the worklist approach to handle traversing a CFG. Using a similar approach used in interpreters it analyzes all of the statements and expressions represented by the vertices in the CFGs and reports any errors found.

The tool was used to scan several PHP applications as well as several WordPress plugins using the implemented modeling of frameworks. The use of models provided speed, efficiency and precision, as the core framework did not need to be analyzed. The analysis resulted in finding 66 new confirmed vulnerabilities in WordPress plugins, where 17 of these were stored vulnerabilities.

PREFACE

This master's thesis was written by three students from Aalborg University. The master's thesis is built on the knowledge gained from the pre-specialization project in our previous semester[17]. The purpose of this report is to describe the creation of a PHP security analysis tool and compare the findings against the results from our previous work. The project started on the 2nd of February, 2015 and ended on the 3rd of June, 2015.

This report is intended for an audience with knowledge equivalent to nine semesters of education in computer science or software engineering. Our previous work lays the foundation for this report and provides a better understanding of the problem[17]. It is recommended, but not a requirement, that the reader has read it.

Citations and footnotes are found after the word, sentence or paragraph they reference. Citations are used for formal sources, whereas footnotes are used to provide additional information as well as reference products, programs, news articles etc. The bibliography and the appendices are located at the end of the report.

Information on where to find the source code for Eir can be found in Appendix C.

CONTENTS

CHAPTER 1

INTRODUCTION

In 2014, the Common Vulnerabilities and Exposures (CVE) database contained 1086 Cross-site scripting (XSS) vulnerabilities and 298 SQL injection (SQLi) vulnerabilities [11, 12]. This proves that security in web applications is still a problem that needs to be addressed. Personal information may be at risk of being leaked or a web application may be at risk of being taken over by an attacker if a web application is subject to an attack.

WordPress, Joomla and Drupal is currently the most used Content Management Systems (CMSs)[1]. All CMSs have had security issues reported in 2014[2][3][4]. As an example, WordPress contained a stored XSS vulnerability believed to affect versions ranging from 3.0 to 3.9. The vulnerability enabled anonymous attacks that could take over a website. The update to resolve the issue was categorized as a critical security release[5].

In addition to vulnerabilities in the frameworks themselves, extensions made by other developers can also contain security issues, which can potentially put the whole installation at risk. In 2013, Checkmarx[6] did a security analysis on the top 50 WordPress plugins in June 2013. They found that 20% of all the tested WordPress plugins contained security issues, most of which were common web application vulnerabilities such as XSS and SQLi. All of the vulnerability plugins combined had been downloaded over 8 million times, at the time of the research. Checkmarx also found that 70% of the top 10 E-commerce plugins for WordPress contained

[1] w3techs.com/technologies/overview/content_management/all
[2] developer.joomla.org/security.html
[3] drupal.org/security
[4] cvedetails.com/vulnerability-list/vendor_id-2337/product_id-4096/
[5] wordpress.org/news/2014/11/wordpress-4-0-1/
[6] checkmarx.com

vulnerabilities[2].

Many vulnerabilities are introduced when data is provided through untrusted sources. In security terms, operations providing data are called *sources* and if the data is from an untrusted provider, the source is called an *untrusted source*. Data from an untrusted source is potentially malicious. Operations that are sensitive to malicious input are called *sensitive sinks* or *sinks*. To prevent these types of vulnerabilities, the untrusted data provided through sources should be sanitized or verified to assure it can be trusted. It is the responsibility of a web application developer to sanitize and verify untrusted input.

It can be difficult for a developer to find security issues, even with knowledge of how to prevent them. Therefore, web security analysis tools have been developed to help find security issues and report them to the developer, so that they can be resolved. Both static and dynamic web security analysis tools exist. A dynamic tool analyzes while the application is executed, whereas a static analysis typically works directly on the source code or compiled code.

During this project an analysis tool capable of finding XSS and SQLi vulnerabilities in PHP code was developed. The tool is named Eir and will be referred to as Eir throughout the report. Eir was tested on real world projects as well as a developed test application and test corpus with security issues. The results showed that Eir was able to find more vulnerabilities in the test application and test corpus than similar free tools. Furthermore, Eir was able to find 66 new confirmed vulnerabilities in real world projects.

The rest of this report will be structured as follows: The remainder of this chapter will outline our previous findings and the problems that this project addresses as well as providing brief information about the problems. Chapter 2 will describe existing theories, algorithms and data structures used in analysis tools. Chapter 3 will provide implementation details of the developed analysis tool. Chapter 4 will describe test scenarios in which the tool has been tested as well as the results of the tests. Chapter 5 will evaluate on the project and the developed tool. Lastly, the report finishes with the conclusion in Chapter 6.

1.1 Previous Work

In our previous work six freely available vulnerability analysis tools capable of analyzing web applications written in PHP were tested[17]. Each tool was tested against a corpus of open source PHP applications and WordPress plugins, as well as a custom test application written for this specific purpose.

The tests showed that while the tools were capable of detecting some vulnerabilities, there were many areas that can be improved upon. Generally, the tools had very poor support for detecting stored vulnerabilities, where the malicious payload

is stored on the server. The static tools were not able to track the data through persistent data stores and the dynamic tools had trouble knowing that the malicious data was used later by the application.

Furthermore, only one tool had additional support for analyzing extensions to bigger frameworks, such as WordPress plugins. This meant that some other tools reported a number of false positives in plugins, because they lacked knowledge about framework specific functions and classes. The tool with additional support for analyzing extensions was capable of using a model of the framework when analyzing an extension. Even though the idea of modeling frameworks in order to avoid having to analyze the entire framework when analyzing an extension is good, the tool did not achieve good results in our tests.

Aside from specific vulnerabilities and framework issues, the static tools' biggest issue were lack of knowledge of certain language features and PHP functions and classes. The lack of knowledge caused many vulnerabilities to be missed, as well as a large amount of false positives to be reported.

During our previous work we focused on XSS and SQLi vulnerabilities. The two types of vulnerabilities will be the main focus for this report as well. The following two subsections are taken directly from our previous work[17], and is included here as they explain the vulnerability types in greater detail which is necessary for understanding the work in this report.

1.1.1 Cross-site scripting

XSS is a vulnerability type in which an attacker is able to run malicious code on the client side of web applications. XSS can be hard for users to detect as their browser automatically executes the malicious code. XSS attacks can be categorized as an injection attack because an attacker is able to inject malicious code into the application using user accessible inputs, such as form fields and GET parameters[1, 16]. The malicious code provided by an attacker can be anything that can be executed on the client side, including HTML, JavaScript, Java applets, ActiveX, Flash etc.[1]. The two most common types of XSS are reflected and stored XSS.

Reflected XSS

Reflected XSS (also called non-persistent XSS) is when a user is tricked into sending malicious code to a server, which is then immediately returned to the user[1, 16]. This can be an unsanitized URL parameter shown on the requested page. An example can be seen in Listing 1.1.

Because the parameter `choice` is not sanitized, the code is vulnerable. An unsuspecting user might visit a malicious link such as:
`http://vulnsite/?choice=<script>alert(document.cookie)</script>`,

Listing 1.1: Reflected XSS example

```
1 <?php
2     $choice = $_REQUEST['choice'];
3     echo 'your choice was: '. $choice;
4 ?>
```

which will cause the malicious JavaScript to be executed. In this case it will result in a popup with the user's cookie information.

To prevent XSS vulnerabilities, it is important that every value that can be malicious is sanitized. To do this, functions such as `htmlentities`, which change special characters in a string into the equivalent HTML character entity, can be used. When a character is represented as a HTML character entity, the browsers know that the character is to be shown on the web page and therefore is not causing the browser to parse the page differently. If sanitization with functions such as `htmlentities` cannot be used, it is important that the strings are checked for malicious content before it is inserted as part of the requested page.

Stored XSS

Stored XSS (also called persistent XSS) is when an attacker is able to store a malicious script on a server. The malicious script will continue to affect users until it is removed, e.g. by a server administrator[1, 16]. In comparison to reflected XSS, stored XSS does not necessarily require users being tricked into sending the malicious request.

An example could be if an attacker is able to change his name in a web application and it is not sanitized neither when stored nor when presented. An attacker would then be able to insert a malicious script in his name that will be executed every time the web application show the attacker's name to someone. Example code can be seen in Listing 1.2. In line 3-4, the updated user information is retrieved from a POST request and is stored in a MySQL database in line 6-9. Line 11-13 retrieves the data for all users from the database, and it is then presented to a user in line 15-17. As can be seen, there is no sanitization performed, which means this code is vulnerable to stored XSS.

Preventing stored XSS can be done the same way reflected XSS can be prevented, since it is the same kind of vulnerability. With stored vulnerabilities, there is the possibility to sanitize values before they are stored or before they are printed on a page. What is preferred vary from developer to developer, but a common approach would be to verify potentially malicious input to make sure it is in an expected format before storing values and then sanitize values before they are printed.

Listing 1.2: Stored XSS example

```
<?php
  ⋮
  $id = $_POST['user_id'];
  $fullName = $_POST['name'];
  // Saving name
  $dbConnection = mysqli_connect("localhost", "dbUser", "
    dbPassword", "exampleApp");
  $sqlStatement = "UPDATE users SET name=". $fullName ." WHERE
    user_id=". $id;
  mysqli_query($dbConnection, $sqlStatement);
  mysqli_close($dbConnection);
  ⋮
  $dbConnection = mysqli_connect("localhost", "dbUser", "
    dbPassword", "exampleApp");
  $sqlStatement = "SELECT * FROM users";
  $result = mysql_query($dbConnection, $sqlStatement);
  // Show all users
  while($row = mysqli_fetch_assoc($result)) {
    echo "Name: " . $row["name"] . "<br/>"
    echo "Id: " . $row["id"] . "<br/><br/>";
  }
  ⋮
?>
```

1.1.2 SQL Injections

An SQLi is an attack on applications that change the semantic meaning of an SQL query in the application. SQLi is at the top of Open Web Application Security Project (OWASP)'s top 10 primarily because of how dangerous they can be and how widespread they are as seen in [17, Sec. 2.1].

SQLi is often used to bypass security and access data that is otherwise unavailable to the attacker. Sometimes it is even possible to use this type of attack to change or remove data in the database. An example of code vulnerable to SQLi is given in Listing 1.3.

Listing 1.3: SQL injection vulnerability.

```
1  $sql = mysqli->query("SELECT * FROM users WHERE
2         username='".$_POST["username"]."' AND
3         password='".$_POST["password"]."'");
4  ... // Use the result of $sql to log in user
```

While this code is fully functional and will give the expected result when used correctly, an attacker can exploit it by injecting SQL. This is because there is no sanitization of the username and password, which means that an attacker could inject SQL commands directly into these fields and change the behavior of the SQL query. A way to do this is to insert `' OR '1'='1` as password, which results in the query shown in Listing 1.4. The changed query will return all users from the database.

Listing 1.4: Changed SQL statement.

```
1  SELECT * FROM users
2  WHERE username='' AND password='' OR '1' = '1'
```

In some cases, it is possible to execute multiple queries in one database call. The attacker is then able to give any command to the database, such as creating a new user and deleting all current admin users in order to completely take over the application. All of this is only possible if the web application makes use of SQL query methods that accept multiple queries such as `mysqli_multi_query()`.

SQLi can be prevented by sanitizing input and by using parameterized/prepared queries or stored procedures. `mysqli_real_escape_string()` is an example of a

function, which can be used to sanitize input that is used in MySQL queries. It is also important to always use quotes around parameters in a SQL query. Using parameterized queries are generally recommended as they provide good protection against SQLi. Parameterized queries cannot be used in every situation, since certain parts of an SQL query, such as table names, column names and keywords, cannot be parameterized. In those cases, white listing can often be used, where the untrusted input is used to select between a set of trusted values that will be used in the query. Stored procedures are another way of giving the same security of parameterized queries as long as they are used without dynamic SQL generation. Dynamic SQL generation is when an SQL query is generated given parameters provided by the application. This might create security vulnerabilities if the parameters originate from a user.

1.2 Problem

One of the issues found in our previous work was the lack of a well-made and actively developed static analysis product. Many of the existing open-source tools currently available provide hard to follow code bases, which makes them difficult to develop further. Additionally, some of the tools do not follow common static analysis theory.

Stored vulnerabilities are also a very serious issue in modern web applications and tools capable of detecting them are needed. Since this is very poorly supported in freely available vulnerability scanners, this report will look into the possibility of statically detecting this kind of vulnerability.

Since most popular web application frameworks, such as WordPress, have a lot of focus on security, they have become quite secure. Their biggest security issues come from the framework extensions, whose developers are not necessarily focusing on the security aspects. Vulnerability scanning tools that have support for analyzing extensions would be useful and is one of the things this report will look into.

This report will focus on PHP applications, since this was the language of choice in our previous work[17].

To summarize, this report will look into:

- Creating a static analysis tool that is expandable and follows existing theory
- Statically detecting stored vulnerabilities.
- Analyzing framework extensions, without needing to scan the framework.

1.3 Theory Based Static Analysis

In our previous work, we studied existing free web security analysis tools. In our study, we found that only few existing tools are applying common data flow analysis theory.

The tool RIPS parses PHP code into a program model by using tokenization and performs flow analysis on the parsed code by iterating through the tokens one by one. RIPS analyzes tokens of interest including variables, control flow branches, inclusion of other PHP files etc. and if a potential issue is found it is reported to the user[4].

THAPS applies a taint analysis based on symbolic execution that works directly on an Abstract Syntax Tree (AST). During analysis, THAPS stores the possible values of each variable, as well as the execution path leading to each value. This information is stored in a tree-structure, which is used to report conditions on each vulnerability. The tree structure can potentially grow very large since it represents all execution paths through an application[7].

The tool Pixy is another static analysis tool, which parses PHP code using JFlex[7] and Java parser Cup[8]. Hereafter, Control Flow Graphs (CFGs) are generated and the tool performs a data flow analysis, which tracks tainted values throughout the program flow[9]. Understanding the workings of Pixy is easy as it is closely tied to common static analysis theory. Pixy is therefore a good candidate for a tool that employs theory and would be a good candidate for developing further. However, the project has not been updated in recent years and does not support features added to PHP since, such as classes and interfaces in PHP. This means that to extend the tool, a new PHP parser has to be developed using JFlex and Java parser Cup in order to parse the new language constructs. After the parser has been developed the tool itself would also have to be updated to support the new parser and new language constructs. All of this would be very time consuming to do, which makes it hard to further develop Pixy.

Based on our experience with looking through the codebases of the tools above, it is our understanding that knowing the theory behind the tool makes it much easier to understand the workings of the tool. Furthermore, given that the theory of static analysis is a much researched area, much of the known theory has been tested and shown to work.

[7] `jflex.de`
[8] `www2.cs.tum.edu/projects/cup`

1.4 Extensibility and Code Quality

By inspecting the source code of the available open source tools, it is clear that, generally, they are not implemented with extensibility in mind. Therefore, it would be a time consuming process to learn how the code works and begin to develop them further.

Since existing tools were not easy to extend it was chosen to develop a new tool. While applying known theory helps make the workings of a tool understandable, it is not enough to make it extensible. In order to be extensible, the code must follow good practices in regards to common quality attributes such as modularity, re-usability and readability.

To ensure the quality and functionality of the code, it is important to have a test suite that test the functionality of the code. The tests work as a safety precaution when making changes to the code and make early detection of errors possible. Tests also work as a sort of documentation of how the code is expected to work. If the tool is to be developed further in the future, the tests will be a valuable asset for the developers. Tests are seen as a benefit in regards to other tools, where we have seen very little to no tests. In addition, the test suite can be used as a measure for the tools stability and correctness if the test cases are chosen wisely.

1.5 Stored Vulnerabilities

As shown in our previous work[17], there are currently no free static vulnerability scanners capable of detecting stored vulnerabilities.

Current static analysis tools typically find vulnerabilities by tracking data flow through an application, to see if data from an untrusted source is used in a sensitive sink, without having been sanitized. In order to detect stored vulnerabilities a static tool has to track data that is being stored in a way such that the data is available across multiple requests to the server. Because of this, a tool will have to track two or more data flows in order to detect stored vulnerabilities. First, there is the data flow where possibly tainted data is stored in a persistent data storage location, such as a database. Then there is the data flow where the data is retrieved from the storage location and used in a sensitive sink.

1.6 Framework Extensions

Framework extensions such as Drupal modules or WordPress plugins are typically developed by third parties, which means that developers with no knowledge of web security can also implement extensions to these large frameworks systems. Being able to analyze third party extensions for security issues is important as these may

compromise not only the extension but also the whole web application. If an extension compromises an entire web application, the security built into the framework becomes irrelevant.

Analyzing extensions for web applications may include function calls that are in the core system of the web application, and not located within the extension itself. Therefore, knowledge of these functions and their purpose during the analysis may be of importance for the analysis result, as the analysis may be incomplete or produce many false positives otherwise. In our study of web security analysis tools we found that in most cases, the only way to get knowledge of the framework while analyzing extensions is by analyzing the framework together with the extension[17]. However, analyzing frameworks is typically time consuming because of the size of the frameworks. Analyzing the framework along with an extension might also cause the analysis tool to report information from the framework, which is not relevant to the analysis of the extension. The analysis tool THAPS introduced the concept of framework models to solve this issue. A framework model can include relevant knowledge about the framework. The model can then be used when analyzing a framework extension, eliminating the need to analyze the framework also.

CHAPTER 2

PRELIMINARIES

This chapter will provide insight into some of the theories and algorithms that are essential in order to understand the inner workings of Eir. In Section 2.1, the CFG data structure will be presented as it can be used to perform data flow analysis. In Section 2.2, taint analysis theory is presented, as theory is essential for an implementation. Section 2.3 will discuss stored vulnerabilities and why they can be problematic for an analysis tool. Lastly, Section 2.4 will provide insight on problems with function whitelisting as used in current tools. Section 2.4 will also discuss frameworks with extensions and why they can be problematic in an analysis.

2.1 Control Flow Graph

CFGs are an essential part of many static analysis tools and an essential part of data flow analysis theory[10, chapter 2]. CFGs are therefore worth looking into when implementing a static analysis tool for detecting vulnerabilities in PHP applications.

A CFG is a graph with vertices, herein referred to as blocks, and edges representing flow. Together it represents all potential control flows in a piece of code and can be used to track how code can be run and the different execution paths that the program can go through. An example of a program can be seen in Figure 2.1, which is a CFG of the code in Listing 2.1. The graph shows the routes through both the branches of the if statement (indicated by T for `true`) and the else statement (indicated by F for `false`).

Once a CFG is created it is possible to perform data flow analysis using well-known algorithms for traversal and variable tracking. A more in-depth look at the applied theories is given in Section 2.2.

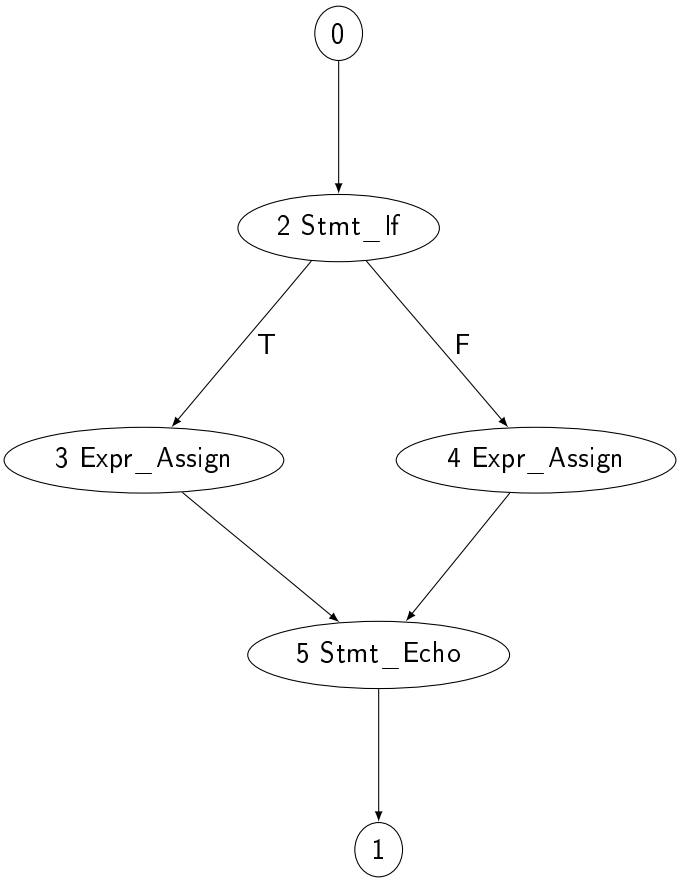

Figure 2.1: Control Flow Graph for code in Listing 2.1

Listing 2.1: If/else PHP script

```
1 if(true) {
2     $var = 1;
3 } else {
4     $var = 2;
5 }
6 echo $var;
```

2.2 Taint Analysis

Tracking taint statically through an applications code is a kind of data flow analysis. Data flow analysis has been studied and used for many years to statically reason about application code. This has resulted in many different types of analysis, such as reaching definition analysis, liveness analysis and available expression analysis[10, chapter 2].

A fundamental and classical way of performing data flow analysis is based on traversal and data tracking through CFGs. What kind of data is tracked through the CFG depends on what the goal of the analysis is. Liveness analysis tracks whether a variables value will be used before the value is overwritten or the program exits, while reaching definition analysis tracks which assignments might have defined a specific variable at a specific point in the application[10, chapter 2].

A traditional data flow analysis algorithm will traverse the CFGs by following the edges and generate `KILL`, `GEN`, `IN` and `OUT` sets of the data being tracked for each block in the CFG. `KILL` and `GEN` sets contain information about data being removed (or *killed*) and generated in a single CFG block. `IN` and `OUT` sets contain data coming into and out of a block, respectively[10, chapter 2]. In taint analysis the data being tracked would be taint information of all variables, which can be represented as a set of different taint tags for each variable.

Listing 2.2: Simple PHP program

```
1 $x = $_GET['a'];
2 $y = (int)$x;
3 while ($y > 0) {
4   $y = $_GET['b'];
5 }
6 echo $y . $x;
```

In order to perform data flow analysis all blocks get an `IN` and an `OUT` set. These sets are initialized to the default values for the analysis being performed. While traversing through the graph the `IN` and `OUT` sets are calculated based on the current knowledge about a given block. In taint analysis, the `IN` set represent the taint values for all variables when the execution reaches that block. The `OUT` set of a block is the `IN` set modified according to the changes that happens within that block. Consider the simple code example in Listing 2.2 containing some simple assignments, a loop and an echo statement. The CFG for this code can be seen in Figure 2.2. The `IN` set in the CFG in Figure 2.2 for block 2 would not have the variable `$x` tainted. After the analysis, the `OUT` set of block 2 would contain a tainted `$x`, since variable `$x` is being assigned untrusted data in that block.

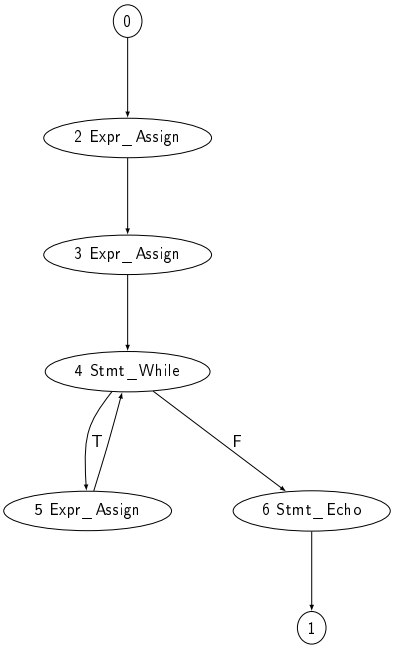

Figure 2.2: Control Flow Graph for code in Listing 2.2

The changes happening within a single block is calculated when the traversal visits that block and is represented in `KILL` and `GEN` sets. The `GEN` set represents the information generated by the code in a block and the `KILL` set represents the data that will no longer be available after this block (or "killed" data). `KILL` and `GEN` is calculated based on the `IN` set of a block and the `OUT` set is calculated based on the `IN`, `GEN` and `KILL` sets[10, chapter 3]. In taint analysis the `IN` set would contain taint information about all variables when entering the block. The `GEN` set contain the taint values being introduced in a block. The `KILL` set contain the taint values being removed in a block, while the `OUT` set is then the `IN` set without the elements in the `KILL` set and with the `GEN` set added. All of this can be represented as formulas like this:

$$IN = \bigcup_{l' \rightsquigarrow l} OUT(l')$$
$$OUT = (IN(l) \backslash KILL(l)) \bigcup GEN(l)$$

Here, l is the block being analyzed. $l' \rightsquigarrow l$ represents all blocks that have an outgoing edge to l.

As can be seen in Figure 2.2 a CFG is not necessarily acyclic. When traversing a graph containing cycles it is important to avoid infinite loops. Data flow analysis will typically continue iterating cycles until it reaches a fixpoint in the `IN` and `OUT` sets. A fixpoint has been reached when there are no longer any changes to the sets.

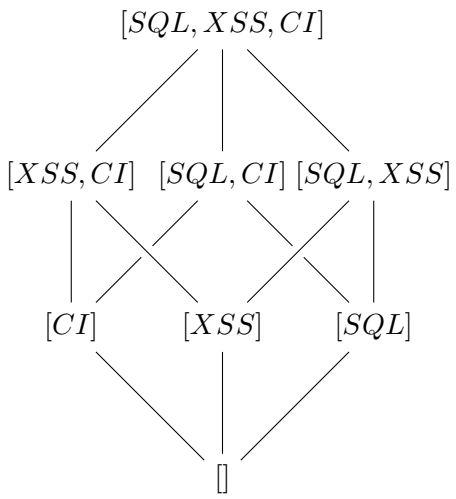

Figure 2.3: Hasse diagram of possible sets of taint tags.

To guarantee that a fixpoint will be reached, the data being tracked needs to be structured correctly. The values being tracked in a data flow analysis are typically values in a lattice. A lattice is a partially ordered set with a greatest lower bound and least upper bound. A lattice is a set with a binary relation, such that some elements are ordered in regard to the binary relation[10, chapter 3]. As an example, a taint analysis might represent a variables taint using a set of taint tags. For the purpose of this example, these tags could be `XSS`, `SQL` and `CI` (Command Injection). Since the analysis is working with sets, the binary relation could be $\subseteq$. The Hasse diagram in Figure 2.3 shows the possible combinations of tags. The lines in the diagram shows the sets satisfying the binary relation. Each variable in a program will be tied with a set of taint tags.

To make sure that the analysis reaches a fixpoint the analysis should only ever add taint when analyzing a single block. This means that after the analysis of a single block in the CFG, the taint set of each variable in the `OUT` set of that block remains the same as it was or taint have been added. A function following this rule is also called a monotone function[14, chapter 2]. By combining the finite Partially Ordered Set (poset) and guarantee that the taint in the `IN` and `OUT` set will only increase, it is clear that in the worst case all variables will have a "full" taint set. This also makes it clear that the analysis will always reach a fixpoint.

As an example, when the analysis enters the while-loop in Listing 2.2, `$y` is not marked as tainted. After the first analysis of the loop body, `$y` is marked as completely tainted, since it has been assigned unsanitized data from an untrusted source. Since the taint status of `$y` has changed, the loop body will be analyzed

again to see if the change in taint could change the taint status of other variables. Since that is not the case, the taint sets of all variables remain the same and a fixpoint has been reached.

2.3 Stored Vulnerability Detection

As briefly described in Section 1.5, stored vulnerabilities are a problem that can be hard to detect. Stored vulnerabilities are split into two or more phases. The first phase occurs when input from a untrusted source reaches a storage location where data can persist through multiple sessions, such as a database. The second phase happens when the tainted data is retrieved from the database and used in a sensitive sink, it results in a stored vulnerability.

A normal scenario of this is where a user provides input to an application, which is then stored on the server. An example of this could be a comment in WordPress, which is stored and later extracted when other users read the comment. If the input enters the database without sanitization and is later extracted and shown to a user without being sanitized, it is possible to XSS attack any user that views the comment.

Sanitization of stored vulnerabilities can occur before the tainted input is inserted into the database, as well as after it is pulled from the database before it reaches a sensitive sink.

In order to find a stored vulnerability it is necessary to detect exactly which storage location is used. In a database, this would include the table and columns the tainted input goes into. Detecting storage locations statically can be difficult when dynamic values are used to determine where input is stored. This could be a dynamically calculated database table. The less accuracy there is in detecting if it is the same data going in and out, the higher the chance of false positives and false negatives.

Stored vulnerabilities are generally more difficult to detect than reflected vulnerabilities, because there are multiple data flows involved before a vulnerability can actually take effect. In order for a static analysis tool to detect data flow through storage locations, it must scan all involved data flows and see if tainted data reaches a sensitive sink. Doing so could require running and saving the results of a taint analysis on both data flows and then afterwards check if there is a vulnerability. This approach would differ from normal taint tracking, as it requires the analysis to save the results of two data flows and then pair them, instead of immediately reporting a vulnerability as soon as tainted input reaches a sensitive sink.

2.4 Function Specification

For a static analysis tool to be able to find vulnerabilities in a web application the analysis has to be aware of the functionality the language provides. In regards to implementing a taint analysis, the analysis needs to be aware of what sources, sinks and sanitizer functions exist.

In our previous work, we tested existing tools for finding web application vulnerabilities. The existing static analysis tools we evaluated in our previous work use a list of names of sources, sanitizers and sinks to find vulnerabilities. An example of how THAPS specifies functions can be seen in Listing 2.3. The exact same approach is used by RIPS. However, security issues may be dependent on sanitizers where a list of names is inadequate, since the result of many functions are dependent on the parameters given to the function. As an example, the `print_r`[1] function in PHP will print the first argument when called and should therefore be regarded as a sink. However, if the second argument is `true` the function will return the string rather than print it, in which case it is no longer considered a sensitive sink.

Listing 2.3: THAPS function specification.[2]

```
1 $SECURING_XSS=array(
2     'htmlentities',
3     'htmlspecialchars'
4     ...
```

Another example is the `htmlentities`[3] function, where the result is dependent on the second argument to the function. The second argument is an optional value, indicating to which degree the string should be sanitized. `htmlentities` will not sanitize single quotes, unless `ENT_QUOTES` is supplied. Escaping of single quotes can be important if the string is to be printed in for example a JavaScript context. If the data is printed within single quotes, it is still vulnerable to XSS. An example where escaping single quotes is important is shown in Listing 2.4. In the example, a text string is retrieved using PHP and printed in the JavaScript. The user input is sanitized using `htmlentities` but single quotes are ignored, making the code vulnerable. An exploit to this could be setting the `userInput` variable to:
`test'; alert('132');//`.

[1] https://php.net/print_r

[2] bitbucket.org/heinep/thaps/src/d22e625361393f53eea8677b6d3d42e0dcfc14aa/scanner/Lib/VulnerabilitySourcesAndSinks.php

[3] https://php.net/htmlentities

Listing 2.4: PHP print in JavaScript context

```
<script>
function trySomething(parameter1) {
    ...
}

var input='<?php echo htmlentities($_GET["userInput"]); ?>';
trySomething(input);
</script>
```

PHP5 introduced a function named `filter_var`[4], where the first parameter is the data string and the second parameter is an integer flag indicating the degree of either filtering or validation to be applied.

A normal whitelist cannot be used to capture the functionality of `filter_var`, since it does not sanitize for SQLi nor XSS by default. However, depending on the flag given as the second parameter, the function can sanitize for XSS, SQLi or both. The `filter_var` function can also be used with verification constants, where the data is only verified but not changed. The return type of the function is dependent on whether the sanitization is successful or not. If successful, the sanitized data string is returned. If the sanitization fails, then `false` is returned. The function `filter_input`[5] can be used in the same way, but also includes a parameter to indicate the type of the returned data.

When specifying functions it is important to include functions that conditionally change the taint value of a variable. In PHP, functions to verify the type of a variable, such as `is_int` and `is_float`, exist. These functions check whether the given input is conforming to the expected type. However, in regards to taint analysis, these can be used to verify that input is safe. In case `is_int` returns `true` the input is an integer and can therefore safely be used in certain sensitive operations. An example of this can be seen in Listing 2.5. In the example the `true` branch is printing out a string concatenated with a value given by the user. This could potentially be a security issue, but as the type is verified to be an integer type, it can only be a valid number, which cannot create a security issue in regards to XSS. These functions have to be specified as influencing the taint status of the input when used in conditions. In the remainder of this report these will be referred to as conditional sanitizers.

[4] php.net/manual/en/function.filter-var.php
[5] php.net/manual/en/function.filter-input.php

Listing 2.5: PHP print in JavaScript context

```
1 if(is_int($_GET['number'])) {
2   echo "The selected number was: " . $_GET['number'];
3 }
4 else {
5   echo "That was not a valid number. Try again"
6 }
```

2.5 Frameworks and Models

In Section 1.6, framework extensions were discussed as a problem for a web application security analysis tool. An analysis tool capable of analyzing a framework extension without having to analyze the framework core is beneficial in regards to the time spent. Irrelevant information about the framework can also be excluded, thereby only giving users relevant feedback. One way to include relevant framework information in an analysis is to extend the function specifications with knowledge of the framework core functions.

Specifying a list of framework functions might not be sufficient in order to correctly model the framework functionality. In event-based systems extension developers can typically define functions that are to be called when an event is triggered. The actual function call might be placed in the framework, and not in the extension.

Additionally, a framework might provide object instances that can be used by the extension. An example of this is the `wpdb` class in WordPress, which is used to interact with the database. WordPress instantiates an instance of the class and assign it to a global variable called `$wpdb`. An extension can the use this instance, when it requires database access. To handle such cases, the analysis will have to know about the available global variables as well as their types, in order to perform a proper analysis.

Therefore, an analysis of extensions with a model of the framework has to be more context-aware when analyzing event-based systems.

CHAPTER 3
IMPLEMENTATION

This chapter will discuss how Eir has been implemented as well as discuss and elaborate on the theory that has been used in the implementation. It provides an overview of the different steps Eir goes through in order to perform a complete analysis of a PHP project. In Section 3.1, an overview of the steps of Eir is provided. Some of these steps are explained in further detail. In Section 3.2, the CFG creation is described and the CFG pruning step is described in Section 3.3. Section 3.4 provides details on how the data flow analysis and its subparts have been implemented. Section 3.5 describes how the analysis of individual blocks in the CFG is handled. Section 3.6 discusses the implementation of function specifications. Section 3.7 provides details on our approach to find stored vulnerabilities and how the approach has been implemented. Lastly, Section 3.8 describes how the tool can be expanded with plugins and how the analysis makes use of the plugins.

3.1 Implementation Structure

As Eir was created to be extensible, the tool was split into parts that can be used separately or as a whole. Eir's process is split into several steps as follows:

1. Parse all files into ASTs

A typical PHP project consists of multiple PHP files, all of which need to be analyzed in order to detect as many vulnerabilities as possible. Therefore, all relevant files have to be discovered and needs to be parsed into an appropriate format for an analysis. Therefore, all PHP files in a project are found and parsed into ASTs.

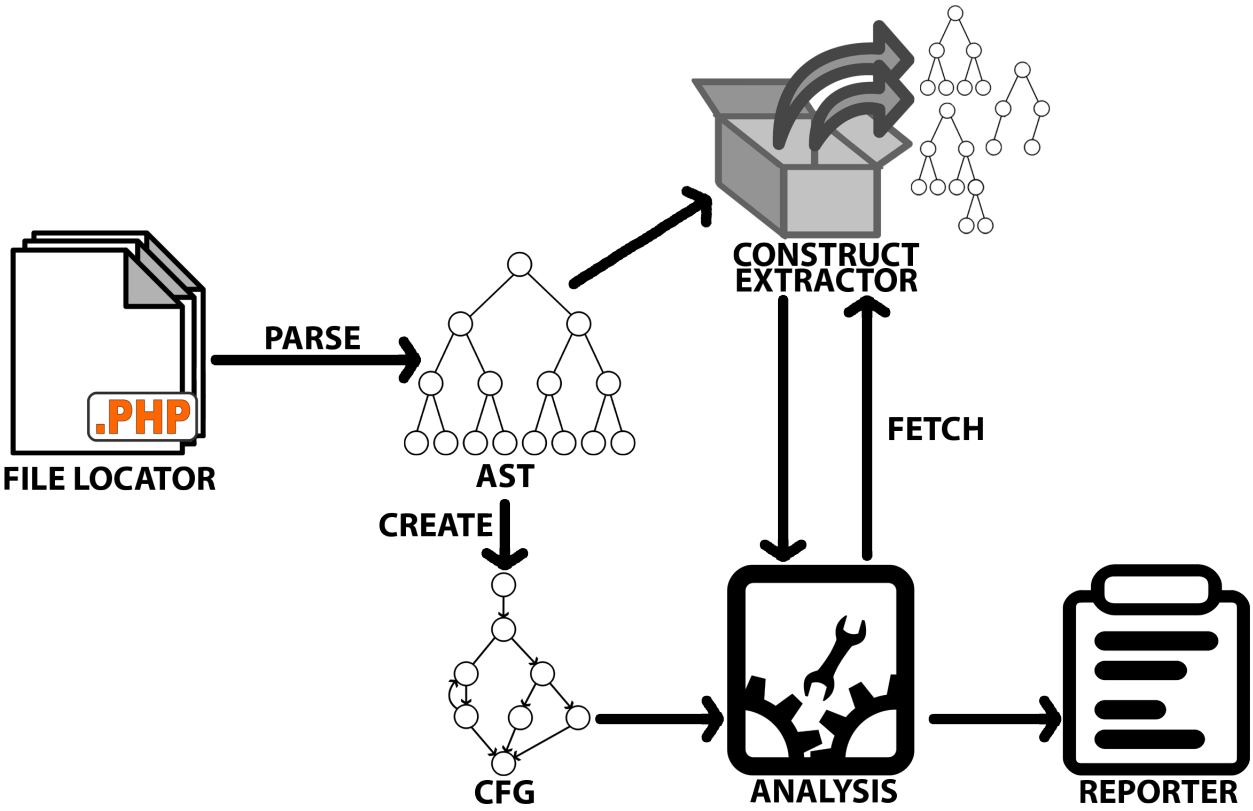

Figure 3.1: Overview of the anaylsis tool

2. Collect basic information

When all files have been parsed into ASTs, they are traversed and basic information about classes, functions, methods etc. is extracted. This information is used later when performing the actual vulnerability analysis.

3. Create CFGs

Every file's AST is traversed in order to create CFGs representing the control flow in the code. This provides CFGs for PHP files, functions, methods and closures in the project.

4. CFG Pruning

In order to remove unnecessary blocks in the CFGs, they are pruned before the analysis begins.

5. Perform forward taint tracking on CFGs

All CFGs that are representing files are marked as a potential entry point and will be a starting point for the analysis. A forward taint analysis is then performed on each marked CFG, while function and method CFGs are traversed as they are used in the application. The traversal also keeps track of potential SQL query strings, in order to be able to track tainted data through the database, which is used for detecting stored vulnerabilities.

5.A Report found vulnerabilities

During the analysis, non-stored vulnerabilities are reported when discovered.

5.B Store information about potential stored vulnerabilities

During the analysis, whenever data is stored or whenever data retrieved from a storage location reaches a sensitive sink, information about the data flow and the storage location is stored.

6. Use result from taint tracking to find stored vulnerabilities

At the end of the analysis, all relevant data flows found during the analysis are paired in order to see if there are any stored vulnerabilities.

3.1.1 Implementation

The implementation is split into modules where each module is responsible for one of the above steps.

The first step is handled by a file parser module that takes a file or a folder as input. If a single file is given, an AST is built. If a folder is given the tool will traverse the directory to find PHP files, and hereafter parse the found files into ASTs. The second step is to collect basic information before starting the analysis. This step is handled by an extractor, which discover classes, functions, methods and closures. This module is working directly on the ASTs before creating CFGs. The third step is the CFG creator module that takes an AST and builds a CFG. When a CFG has been created it is given as input to a pruner, which will remove unreachable blocks and redundant intermediate blocks.

When the CFG creator and the CFG pruner has finished, the CFG is used in the taint analysis module. In the implemented tool, this is a forward taint tracking analysis. This is handled by traversing the CFG(s) and whenever a block is visited, the block is analyzed for changes in the taint. The analysis uses an inclusion resolving module, which tries to discover included files, so they can be included in the analysis.

The taint analysis provides a reporting module with data about taint reaching a sink and then lets the reporting tool decide if a vulnerability should be reported. This also includes stored vulnerabilities. However, due to the fact that stored vulnerabilities involve multiple data flows, the vulnerabilities are reported in the end. They are stored as potential vulnerabilities until it can be determined whether a vulnerability is present or not.

Each of these separate sections of code have their own end to end tests that cover a lot of the code in order to make it easy to make modifications to the code with assurance that the changes do not break existing functionality. At the time of writing, the project has more than 250 tests.

3.2 CFG Creation

In the beginning of the analysis a CFG is created for each individual file. CFGs for methods and functions are created separately when they are needed. In order to create a CFG, a generator was needed. As we have not been able to find a suitable CFG creator for PHP code, one was created. The CFG creator iterates through an AST and builds a CFG.

The CFG creator takes an AST created by PHP-Parser by nikic[15], which is exported from a PHP script into XML format, which is then importet into Eir and used by the CFG creator. The CFG is created so that each vertex contains the AST of the statement it represents. This AST allow the analysis to analyze the statement each block represent. The edges created in the CFG are also marked with a type to identify true and false branches of conditionals that can be used in later analysis. The CFG creator was created to be reusable and easy to use in order to make it possible to use it for different purposes.

Since the PHP parser provides an AST, the tool creates a CFG from the AST by utilizing a visitor pattern. Using a depth first strategy, every node is visited and whenever a node of interest for the control flow is visited, an appropriate handler for that node is called. The CFG creator generates a CFG representing the possible flow paths in a piece of code. The generator does not try to resolve function and method calls in order to connect multiple CFGs.

The CFG creator is not complete, as parts of the PHP language are not supported because of limitations in time. One of the left out features is PHP's `goto` statement. It was deemed reasonable to give less priority to `goto`, since it is one of the rarer used features in PHP. This can be seen in a study by Hills et al. [5], where they detected no `goto` usage in a corpus of 19 systems. To back this up even further, a module to count `goto` statements in WordPress plugins was developed for Eir. Counting goto statements in 40.557 WordPress plugins from the official WordPress

repository[1] only 20 plugins was found using `goto`.

3.3 CFG Pruning

In order to remove unnecessary blocks in the CFGs they are pruned using a simple pruning algorithm.

The CFG building algorithm includes all statements in the code, even if they are obviously unreachable. The first step of the pruning algorithm removes unreachable blocks. The algorithm works by traversing the graph from the root block and marks all blocks that can be reached from the root block. When the graph has been traversed, the blocks that have not been visited are definitely unreachable and are removed from the graph.

Secondly, the pruner removes redundant intermediate blocks that have been inserted to ease the creation of the CFG. These intermediate blocks do not represent any code and can always be safely removed. These blocks are detected by a simple iterative search and removed after moving incoming edges to the correct child nodes.

An unpruned CFG is shown in Figure 3.2, which is generated from the code seen in Listing 3.1. The code after the `continue;` statement is clearly unreachable. This can also be seen in the generated CFG, since the following if-statement has no in-edges. Furthermore, it can be seen that some of the blocks are unnecessary, such as block 5, since it does not represent any code statements. Figure 3.3 shows the CFG after it has been pruned.

Listing 3.1: Code that generates example CFGs

```
1 for($i=0; $i<10; $i++)
2 {
3   continue;
4   if(isset($_GET['test']))
5     echo $_GET['test'];
6 }
```

[1] Revision 1148316. Empty projects and projects that could not be downloaded was excluded.

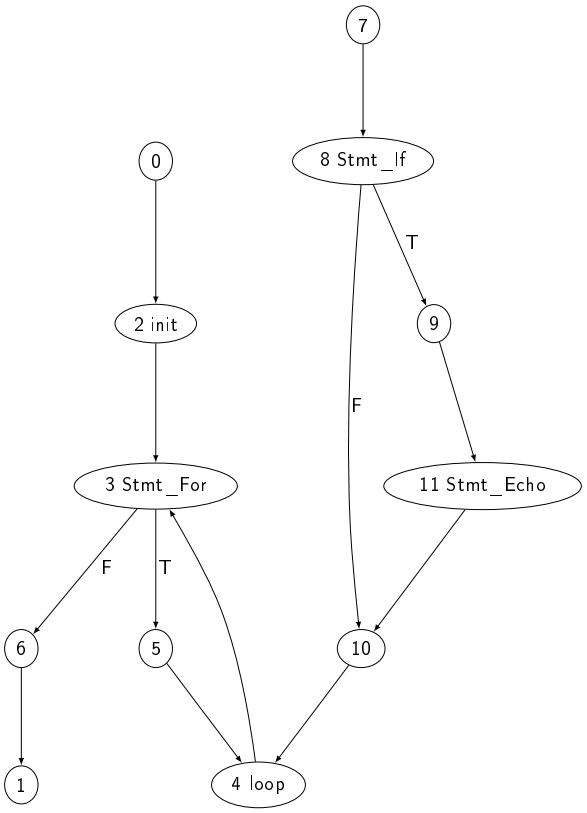

Figure 3.2: Control Flow Graph for Listing 3.1 before pruning step

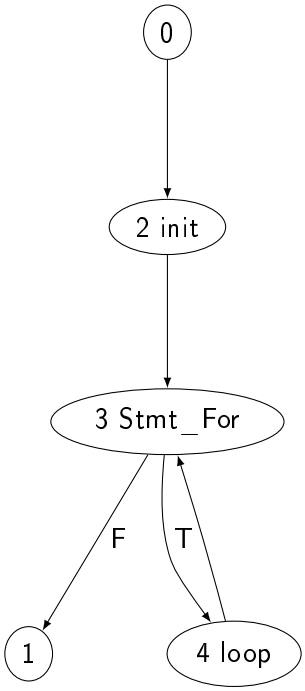

Figure 3.3: Control Flow Graph for Listing 3.1 after pruning step

3.4 Data Flow Analysis

When the CFG for a file has been pruned, the analysis for finding vulnerabilities is applied. The analysis is a data flow analysis, which tracks the taint values of variables throughout a PHP program by analyzing the CFG. Before starting the taint analysis, an initial setup is performed. This includes setting up initial taint statuses, for example marking the PHP superglobals, such as `$_GET` and `$_POST`, as tainted.

The CFG is traversed using a work list approach, where each block in the CFG that need to be analyzed is stored in a work list. This list is then updated whenever a block has been analyzed or need to be reanalyzed. The traverser is implemented so that it is not tied to any specific analysis and it can therefore be used with other types of data flow analyses if needed in the future. The CFG is traversed in a forward manner, meaning that the analysis starts from the CFG's entry block and finishes in the exit block. The traverser can also easily be used with different types of data flow analysis that require other traversal strategies.

The data flow analysis used is a `GEN/KILL` algorithm where the data being tracked is taint tags on variables. Whenever a block is analyzed, taint is either introduced, removed, or remains the same. To do this, each block in the CFG is tied to an IN

and OUT variable storage containing the information and taint known about each variable when entering the block as well as when leaving it.

When the block being analyzed represent a condition in an if-statement or loop construct a special condition analyzer is used. The condition analyzer tries to determine if the condition used can affect the taint values of a variable. As seen in Section 2.4, there are certain functions in PHP that can be used to check if the value of a variable is valid. These types of functions, as well as comparisons with hardcoded values, are often used as conditions and can cause a variables taint to differ depending on which outgoing edge is taken from the condition block. A few examples of conditions where the condition analysis changes the taint value of one branch can be seen in Listing 3.2.

Listing 3.2: Examples of conditional sanitization.

```
1 if ($var === "hardcoded") { .. // $var safe in true branch.
2
3 if ($var != "hardcoded") { .. // $var safe in false branch.
4
5 if (is_int($var)) { .. // $var safe in true branch.
6
7 if (!!!is_numeric($var)) { ... // $var safe in false branch.
```

3.5 Taint Analysis and Taint Tracking

Each block in the CFG contains the AST of the statement it represents. Whenever a block is visited by the analysis, that AST is analyzed in order to determine if taint is being introduced or removed. To analyze the AST the analysis iterates through the AST using the same approach as a typical interpreter. It performs a bottom up analysis, where taint is propagated from leaf nodes of the AST toward the root. This allow the analysis to precisely determine what happens in the code and how it affects taint. The result of the AST analysis can then be used as the `OUT` set of the CFG block.

If the analysis sees a function or method invocation, all arguments are analyzed to determine their taint values. The invocation target is then looked up to see if it is a source, sink or sanitizer. If it is a source or a sanitizer the correct result taint is determined and propagated further in the analysis. If it is a sink, the taint values of the arguments are used to determine if a vulnerability should be reported. If the function represents a storage location and the arguments are tainted, the relevant information about where the tainted values are stored are saved until the end of the analysis. Handling of stored vulnerabilities is explained in Section

3.7. If the invocation is to a function or method implemented in the project being analyzed, the function or method is retrieved, parsed and analyzed. Every time an invocation happens, the function or method is analyzed. This is also called polyvariant analysis. Sometimes the analysis is unable to determine what function is called, such as if the code uses variable functions or has not been modeled in the function specifications. An example of a variable function call can be seen in Listing 3.3 where the `get_name_of_func` is a function returning a string value containing the name of a function, which is then called in the next line. If the analysis cannot resolve the function being called, it assumes that it will return a value with taint corresponding to the taint of the arguments given.

Listing 3.3: Variable function call.

```
1 $myFunc = get_name_of_func();
2 $myFunc();
```

Whenever the analysis sees an include statement, it will try to resolve the include such that the included file can be analyzed. As seen in our previous work, dynamic includes in PHP can be hard for static analysis tools to handle[17]. When including files in PHP it is very common to concatenate a hardcoded filename to a calculated path. Listing 3.4 contain a few examples of this pattern of includes from WordPress. To handle this, Eir will look for the hardcoded string values in the include statements and try to match that string with the filenames in the project. If a matching file can be uniquely identified, the file will be analyzed. Otherwise, the analysis considers the include unresolvable and continues with the analysis.

Listing 3.4: Common include statements as seen in WordPress v.3.9.2

```
1 require( dirname( __FILE__ ) . '/wp-blog-header.php' );
2 require_once ABSPATH . 'wp-includes/class-phpass.php';
3 include_once(ABSPATH . WPINC . '/class-wp-xmlrpc-server.php');
```

Since Eir uses common data flow analysis theory, loops are modelled in the CFGs and the number of iterations in a loop is based on the variables taint values. To handle loops such as `foreach` loops, the analysis will go through every known variable in the array being iterated and take the worst case scenario of taint values for both the dimension and value variable specified in the loop. This way of handling foreach loops allow the analysis to assume a worst case taint value for the iteration variables. An example of this can be seen in Listing 3.5, where a single element in the array is tainted, but the other is not. When the analysis reaches the `foreach`

loop, the `$value` variable will get the combined taint of all known variables in the array. In this case it means that `$value` is completely tainted and a vulnerability will be reported in the `echo` statement.

Listing 3.5: Standard PHP `foreach` loop.

```
1 $array = array(1 => $_GET["tainted"], 2 => "not tainted" );
2
3 foreach($array as $key => $value) {
4   echo $key . " : " . $value;
5 }
```

While this technique works for many scenarios, it is not without its issues. Some applications use `foreach` loops to change each element in an array. An example of this can be seen in Listing 3.6, where every element in the loop is sanitized by casting the values to integers. To handle such scenarios the analysis would have to detect that dimension variable `$key` is used to access the array and therefore every element in the array should be sanitized. This would require the analysis to keep track of extra information during the analysis, which Eir is currently not able to do.

Listing 3.6: Problematic foreach loop.

```
1 foreach($array as $key => $value) {
2   $array[$key] = (int)$value;
3 }
```

3.5.1 Variable Representation

In order to track tainted variables through an application, it is important to model the variables and their scope correctly. The scope of a variable is relevant, since the analysis needs to know what variables are accessible in a given context. In Eir, we distinguish between five different scopes. First there are PHP's superglobals, such as `$_GET` and `$_POST`. The superglobals are accessible everywhere in an application. There are only a few superglobals in PHP and it is not possible for applications to create their own. It is possible for an application to create standard global variables, which are variables defined outside any function or class. Global variables can be used directly by code executing in a global scope. To use a global variable in a function, it needs to be specified with the `global` keyword[2]. Variables declared in

[2] php.net/manual/en/language.variables.scope.php

a function or method are local variables and are only accessible within the function/method where they are declared. To make sure that the analysis knows where to look up variables, the analysis keeps track of whether it is currently analyzing code in the global scope or analyzing a function or method. This makes it possible for the analysis to lookup the right variables during analysis.

The final two variable scopes are static variables/properties and instance variables. Variables marked with `static` are variables that are initialized once and then exist for the remainder of the execution. A common example of this is a static class property, which is a static variable defined in a class that can be accessed anywhere by using the class and variable name. Finally, there are instance variables, which are non-static variables defined in a class, meaning every instance of a class has its own set of instance variables. Instance variables can only be accessed by going through the class instance.

Arrays are a powerful data structure in PHP, which is used extensively in many applications. Arrays can be used as normal arrays known from other languages such as C# and Java where integers are used as indexes into the arrays. However, PHP also permit using arrays as a map, allowing other types, such as strings, to be used to lookup values instead of integers. Because arrays can contain many different types of data, including other arrays, array elements are also regarded as instance variables.

In order to handle nested arrays correctly we were inspired by the way it is done in Pixy. Pixy model arrays as trees, where each edge represents a dimension and each node represents an actual data value[8]. This means that the array created in Listing 3.7, will be represented as the tree in Figure 3.4. This tree structure is then used whenever an array is accessed in order to figure out what data is accessed. The same basic idea is used for representing classes.

Listing 3.7: Example of array declaration in PHP

```
1 $array = array(
2     "foo" => "bar",
3     42    => 42,
4     "multi" => array(
5           "one" => $a,
6           "two" => "asdf"
7     )
8 );
```

This kind of structure makes it possible to precisely model an array where only some of the internal elements are tainted. An example of such an array would be the `$_SERVER` superglobal, where some elements, such as `HTTP_USER_AGENT`, can be

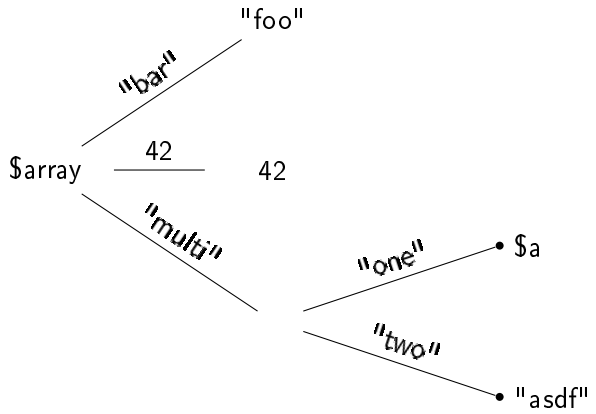

Figure 3.4: Tree structure for array in Listing 3.7

changed by a user, while others are defined by the server, such as `SERVER_ADDR`. A similar approach is used when handling classes and their fields.

3.6 Function Specification

As stated in Section 2.4, it is essential that a taint analysis is aware of the various sources, sinks and sanitizers in a language and various frameworks. Functions such as PHP's `mysqli_real_escape_string`[3] are important since they have a direct effect on the taint status of data. This can be seen in our previous work, where THAPS and Pixy failed to detect vulnerabilities since they did not know about the MySQL Improved library in PHP[17].

As shown in Section 2.4, a simple list is not always sufficient to capture the context in which the functionality is used. Therefore, a JSON formatted specification has been developed. The specification format is developed to support parameter dependent analysis. The JSON specifications are loaded from files, which makes the JSON specification easy to extend and use in the analysis. Eir loads specifications from all JSON files found in folders specified by the user of the tool. The specification of PHP functions and framework functions can be split up, and thereby framework functions can be excluded by simply removing the folder path with the framework functions specifications in. In Listing 3.8, a specification for `htmlentities` is shown. The first key **name** is the name of the function, which is used when looking up functions from the analysis. The **aliases** is an optional string array of aliases for the function. These aliases are used as a second try when looking up functions, which means the most common name should be used with the **name** key to ensure the most optimal lookup. The **formats** key holds the call formats given as strings.

[3] `php.net/manual/en/mysqli.real-escape-string.php`

Listing 3.8: JSON specification for `htmlentities`

```
{
  "name":"htmlentities",
  "aliases": [ ],
  "formats": [ "htmlentities(string, int, string, bool)" ],
  "totalParameters":4,
  "parameters": [
    { "number":1, "name":"string", "type":"object", "
      isReturnValue":true },
    { "number":2, "name":"flags", "type":"flag", "values": [
        {"value":2, "outputStatus":"XSS_HTML" },
        {"value":3, "outputStatus":"None" },
        {"value":0, "outputStatus":"XSS_ALL" },
        {"value":4, "outputStatus":"None" },
        {"value":8, "outputStatus":"None" },
        {"value":128, "outputStatus":"None" }
      ]
    },
    { "number":3, "name":"encoding", "type":"string" },
    { "number":4, "name":"double_encode", "type":"bool" }
  ],
  "returnType":"string",
  "defaultStatus":"XSS_SQ"
}
```

The **totalParameters** key is an integer stating the number of parameters that the function has including optional parameters.

The **parameters** array is used to specify information for the parameters. A parameter should always include the index (the **number** key), a parameter name for readability and standard PHP functions should be using the same parameter names as the PHP manual. The **type** key is used to handle types accordingly in the analysis. The **isReturnValue** is a Boolean specifying the parameter which is returned. This is used to propagate existing taint values to the return value of the function. Lastly, sanitizers can have a **values** array, which holds taint information for a specific value. In the example in Listing 3.8, the second parameter is a flag type and if the flag value is 3 then the return value is not tainted in regards to XSS. This feature could also be used to whitelist safe regular expressions used in functions such as `preg_match`[4]. The idea of whitelisting regular expressions is also explored in [18]. It should be noted that due to time limitations the analysis is not currently using the values array. The **ReturnType** key is specifying the type of the return value. As the example in 3.8 is a sanitizer example, the specification has a **defaultStatus** key. This is used as a fallback value in case there is no matching value in the parameter array that can provide a taint tag.

All the taint tags used in the JSON specification should be in the matching enum in the C# code. The XSSTaint enum is shown in Listing 3.9.

Listing 3.9: C# enum specifying taint tags

```
1  [Flags]
2  public enum XSSTaint
3  {
4    None = 0,
5    XSS_JS = 1,
6    XSS_HTML = 2,
7    XSS_SQ = 4,
8    XSS_AllQ = 8,
9    XSS_ALL = XSS_JS | XSS_HTML | XSS_SQ | XSS_AllQ
10 }
```

There are slight variations for the JSON specifications as parameter dependency, return values, taint status etc. are not the same for sources, sinks and sanitizers.

Another example is the `print_r` function in PHP. In case the second parameter is `true` the function will return and should not be regarded as a sink. However, if the Boolean is `false` or not specified the function will print out the string given as the first parameter and should thereby be regarded as a sink. In the JSON specification

[4] `php.net/manual/en/function.preg-match.php`

the `print_r` is specified as shown in Listing 3.10, where the key **outputs** in the values array is a Boolean indicating whether the function outputs or not.

Listing 3.10: `print_r` function specification

```
{
  "name":"print_r",
  "formats":[ "print_r(object, bool)", "print_r(object)" ],
  "totalParameters":2,
  "parameters": [
    { "number":1, "type":"object", "name":"expression" },
    { "number":2, "type":"bool", "name":"return", "optional":true
      , "values": [
        {"value":true, "outputs":false },
        {"value":false, "outputs":true }
      ]
    }
  ],
  "defaultOutput":true,
  "returnType":"string",
  "defaultStatus":"XSS_ALL"
}
```

The conditional sanitizers are specified using the same format as normal sanitizers. If the analysis encounters a condition with a conditional sanitizer, the taint status inside the if statement is determined by the conditional sanitizer. In Listing 3.11, the `$var` variable is regarded as untainted in the `true` branch, thus no XSS vulnerability will be reported in line 2. In the `else` branch the `$var` is regarded as tainted, hence a potential XSS vulnerability is present, and an XSS vulnerability is reported in line 4 of the script. If the `is_int` is negated the taint handling of

Listing 3.11: PHP example using `is_int`

```
1 if(is_int($var)) {
2     echo $var;
3 } else {
4     echo $var;
5 }
```

the `true` and `false` branch are flipped such that the `else` branch instead will be regarded as untainted, and in the `true` branch the `$var` is regarded tainted.

3.7 Stored Vulnerability Detection

As mentioned in Section 2.3, in order to detect stored vulnerabilities it is required to track data flow both into a storage location and out of a storage location again.

These need to be tracked individually as the analysis may encounter the retrieval of data before the insertion of data. Because of this, the order in which data is inserted and retrieved is disregarded in the analysis. This is because an exploit is usually stored in one execution and exposed to a different user in another. Because of this behavior our implementation tracks stored vulnerabilities by saving whenever a tainted value is used in a storage insert function. This is kept track of while the analysis continues. The analysis also tracks whenever a value is retrieved from persistent storage. When a value is taken from a storage location, the analysis assumes that it is tainted in regards to both XSS and SQLi. If the value from a storage reaches a sensitive sink without being properly sanitized, it is saved while the analysis continues.

Once the analysis is done traversing the entire CFG and all of the storage related vulnerabilities have been found, the analysis pairs all of the incoming and outgoing taint flows by storage location and checks if any of the pairs are vulnerable both on the incoming and outgoing path. If this is the case, a vulnerability is reported.

An example of this is shown in Listing 3.12. Here the analysis would detect that user input is inserted into the table `products` without being sanitized. It then detects that it is retrieved from the database and assigned to the variable `$result`. It is then given the sensitive sink `echo` without being sanitized. By pairing the ingoing and outgoing data from the table `products` the analysis will report a stored vulnerability as potentially tainted input is able to reach a sensitive sink without being sanitized.

Listing 3.12: Stored vulnerability

```
1 mysql_query("INSERT INTO products (product_name)
2               VALUES (" . $_GET["UserInput"] . ")");
3 $result = mysql_query("SELECT product_name FROM products");
4 echo $result;
```

Storage locations can come from several different places, including databases, files and other locations. This means that implementations for each type of storage location must be present in order to properly handle when an operation is an insertion operation and when it is a retrieve operation. In a database it would be required to detect the database name, table name and column as well as which rows are saved to, in order to maximize precision.

Eir was implemented to support SQL queries and is able to detect `INSERT INTO`,

`UPDATE` and `SELECT` statements on a specific database table. It then tracks taint in incoming and outgoing data from the database and pairs them by table name when checking for vulnerabilities. This means that analyzing the code in Listing 3.12 results in finding a stored vulnerability in the storage location `products`. The results from analyzing the code in Listing 3.12 can be seen in Figure 3.5. Here it is seen that the reporter reports both the point where tainted data is inserted into the database as well as where tainted data reaches a sink. The vulnerability scanner also finds an SQL injection as one is present in the `INSERT INTO` command, as user input is inserted directly into a query without former sanitization.

```
--------------------
SQL vulnerability found on variable: UserInput on line: 3 in file: testswitch.ph
p
Include sequence: testswitch.php
--------------------

--------------------
Stored XSS found - Ingoing: UserInput on line: 3 in file: testswitch.php
Tainted outgoing reaches sensitive sink: result on line: 6 in file: testswitch.p
hp
Include sequence ingoing: testswitch.php
Include sequence outgoing: testswitch.php
--------------------
Time spent: 00:00:00.3890812
Found 2 vulnerabilities.
```

Figure 3.5: Result of stored vulnerability scan

As Eir is only a prototype and proof of concept, the features of stored vulnerability analysis were implemented to overapproximate results and be naive. The tool does not look into queries beyond checking if it is an insertion or retrieval, as well as detecting the table name used. It does not check if there are safe keywords such as `COUNT(*)` in the query, that assure only an integer is returned. It also assumes that all rows and columns in the returned data are tainted. This is also the case when inserting data, where the tool assumes that every column is tainted if tainted data is inserted into the storage.

3.8 Analysis Plugins

When analyzing plugins, the function specification format described in Section 3.6 is only capable of specifying specific sources, sinks and sanitizers. When analyzing framework extensions, it is possible to specify the framework specific functions in this collection. With complex frameworks, this is not always enough to perform a good analysis.

As mentioned in Section 1.6, THAPS introduced the concept of framework models, which made it possible to provide additional, framework specific, knowledge to the analysis. This is a great idea, since it makes it possible for users to modify the analysis so that it can scan extensions to frameworks of their choosing. Since vulnerabilities in framework extensions, such as WordPress plugins, are very common, it was important that Eir also allowed developers to scan framework extensions without having to scan the entire framework along with it. Because of this, the idea of modeling the important elements in frameworks was pursued.

In Eir, the idea of framework models has been extended into a plugin architecture. By making it possible for developers to write plugins that can be used in the analysis, it is possible to extend the analysis even further than just modeling frameworks.

There are many different kinds of PHP applications and the analysis requirements of individual projects and developers differ. This makes it very hard to satisfy everyone using a single solution. By enabling users to write their own plugins that modify certain aspects of the analysis, such as how certain function calls are handled, how much information is collected during the analysis or simply how the analysis reports vulnerabilities, it is possible for users to perform an analysis that fits better with the project they want to analyze.

An example of a plugin can be seen in Appendix B. This plugin is a simple plugin that writes basic vulnerability information into a file, which makes it possible to inspect the vulnerability reports after the analysis is done.

3.8.1 WordPress Modeling

To allow Eir to scan WordPress plugins without having to scan the WordPress framework along with every plugin, a small plugin to Eir was developed, that is able to handle some WordPress specific functionality.

WordPress has a range of sanitization functions that plugins can use, such as `esc_html`[5] and `esc_attr`[6], which were modeled in the JSON function specifications.

On top of this, plugins in WordPress can hook into the framework in order to perform tasks such as creating a settings page in the admin panel. An example of this, from the WP Construction Mode version 1.8[7] plugin, can be seen in Listing 3.13. Here WP Construction Mode is asking WordPress to use a function called `under_construction_menu` to build an admin page. It is very common for WordPress plugins to hardcode the function to be called, when registering the hook. Since Eir by default will only analyze functions and methods that are actually used, this kind of function usage needs special handling, which can be done in an Eir plugin.

[5] `codex.wordpress.org/Function_Reference/esc_html`

[6] `codex.wordpress.org/Function_Reference/esc_attr`

[7] `wordpress.org/plugins/wp-construction-mode`

The Eir plugin can be used when analyzing WordPress plugins. This plugin hook into the analysis and detects when `add_action` or a similar function in WordPress is called. If it detects such a function call, it tries to look up the specified function and then cause the analysis to analyze that function. This makes sure that as long as the function being registered can be resolved (such as if it is a hardcoded string), the function will be analyzed.

Listing 3.13: Adding admin menu in WP Construction Mode 1.8

```
1 add_action('admin_menu', 'under_construction_menu');
```

WordPress also allows plugins to save key-value pairs, called options, in the database using functions such as `get_option`[8] and `update_option`[9]. This is commonly used by WordPress extensions to store plugin settings. WordPress stores the data safely, but does not sanitize the data, making the options a possible storage location for stored vulnerabilities. In order to detect this type of vulnerabilities, the analysis needs to be able to detect the key being used to store and retrieve options. Since the key is typically a hardcoded string, it is possible to use the same approach as with the WordPress hooks. The Eir plugin checks whether an options function is called and then tries to resolve the key being used. If an option is stored and the key is resolvable, the Eir plugin remembers the taint status of the data being stored. If data is retrieved, the Eir plugin marks the output as potentially tainted data, just as it is done by Eir when retrieving data using SQL. Using the Eir plugin it is possible to detect stored vulnerabilities in WordPress extensions, where the options functionality is used as storage location.

Since WordPress automatically escape certain characters in user input before it can be used by a WordPress extension, the Eir plugin overwrite the initial taint status of the superglobal variables. Furthermore, the global `$wpdb` variable instantiated by WordPress is inserted into the variable storage before the analysis begins.

[8] `codex.wordpress.org/Function_Reference/get_option`
[9] `codex.wordpress.org/Function_Reference/update_option`

CHAPTER 4

RESULTS

This chapter will discuss how Eir was tested. Section 4.1 presents the test corpus from our previous work. Section 4.2 provides the results of our tests with Eir. It also summarizes on the test results from our previous work as well as compare these results to the results of Eir. Lastly, Section 4.3 describes additional tests performed with Eir, the additional tests are performed on WordPress plugins and an application developed by a local business herein called BizApp.

4.1 Test Corpus

Testing Eir was done by comparing it to the results of the tools that were tested in our previous work[17]. The test corpus of our previous work can be seen in Table 4.1. This test corpus consist of several open source applications and a test application created to evaluate analysis tools in our previous work. The test application is referred to as Test Application in the following. Furthermore, the test corpus consists

of several WordPress plugins that can be found in Appendix A.

Table 4.1: Test corpus from previous work[17]

	SQLi	sXSS	rXSS	**Total**
Test Application	24	2	60	**86**
MyBB 1.8.1	1	1	1	**3**
MODX Revolution 2.2.14	0	1	1	**2**
osCommerce 2.3.3.4	1	5	2	**8**
TestLink 1.9.11	2	0	3	**5**
WordPress 3.9.2	0	1	0	**1**
WordPress Plugins	4	10	65	**79**
Drupal 7.31	1	0	0	**1**
Total	**33**	**20**	**132**	**185**

Eir was also tested against BizApp, an application created by a real world company. Furthermore, three new plugins not present in the test corpus have been analyzed, as well as updated versions of the plugins present in the test corpus.

4.1.1 Function Summaries

Initial testing revealed that scanning large frameworks with Eir took an unreasonable amount of time. Because of this, support for basic function summaries was added to Eir. When the analysis reaches a function call, the arguments' taint status is calculated. The analysis then looks for a function summary matching the arguments. If a summary is found, the summary is used instead of re-analyzing the function. If a summary is not found, the function is analyzed and a summary is generated, which is stored so it can be used if the function is called again later in the program. A summary contains taint information about a functions result.

Using summaries makes the analysis faster, since functions only have to be analyzed a few times, rather than every time they are called. However, summaries make the analysis less precise. In the current implementation of function summaries, the status of global and superglobal variables are ignored when selecting summaries. This means that if the taint status of a global variable has changed between two calls to the same function, some vulnerabilities in the function might be missed. The summaries are used with all the applications in the test corpus, except Test Application, MODX Revolution and the WordPress plugins, because the large frameworks would take days to analyze without them.

4.1.2 BizApp

BizApp is a commercial tool developed and maintained by a local business. The application is an e-commerce management system created from scratch without use of any frameworks. The application was made with an obvious lack of focus on security and several errors were found manually within a short period of time. BizApp was chosen as an addition to the test corpus as it is a real world example that is not open source. It is a good example to use as it depicts the state of some of the real world applications created by companies with little experience or focus on security.

4.1.3 Additional WordPress Plugins

Since many of the vulnerable WordPress plugins in the test corpus have been patched since the tests in our previous work, Eir was also tested against the newest versions of each of the patched plugins. Furthermore, a few plugins were randomly selected from the featured and popular plugins on the official WordPress plugin page[1].

4.2 Results

The results of analyzing the test corpus can be seen in Table 4.2.

[1] `wordpress.org/plugins` - 2015/06/03

Table 4.2: Results of the Eir analysis on the test corpus

Eir	SQLi			sXSS			rXSS		
	TP	FP	FN	TP	FP	FN	TP	FP	FN
Test Application	20	6	4	2	0	0	50	7	10
MyBB 1.8.1	0	0	1	0	0	1	0	9	1
MODX 2.2.14	0	0	0	0	0	1	1	0	1
osCommerce 2.3.3.4(*)	2	1	0	2	15	4	55	171	0
TestLink 1.9.11	0	0	2	0	0	0	6	5	0
WordPress 3.9.2	0	0	0	0	0	1	0	7	0
WordPress Plugins	7	14	1	19	5	0	67	8	2
Drupal 7.31	0	0	1	0	0	0	0	0	0
Total	29	21	9	23	20	7	179	207	14

TP = True Positives.
FP = False Positives.
* = 117 Unconfirmed stored vulnerabilities.

Eir was tested on the Test Application developed for our previous work. By analyzing the Test Application most of the implemented vulnerabilities were found. The tool reported false positives for both reflected SQLi and XSS vulnerabilities, most of which were cause by regular expressions not being analyzed. One of the false negative SQLi vulnerabilities were caused by not handling the PHP `implode`[2] function. Another false negative SQLi vulnerability was due to the lack of analyzing the context in regards to quotes in SQL statements. Most of the false negative XSS vulnerabilities were caused by not handling the `filter_var` function. An XSS vulnerability was unreported, because user input was used as the key for an element in an array.

4.2.1 Framework Results

In MyBB nine false positive XSS vulnerabilities were reported by Eir. Most of the false positives were reported because of unknown database functions.

In the MODX Revolution CMS system one reflected XSS vulnerability was found. The reported vulnerability is a true positive, even though it is located in a file that is removed after MODX Revolution's default installation.

In osCommerce, Eir reported 230 vulnerabilities, where 171 are false positive XSS vulnerabilities. These are reported because of a function named `tep_output_string` in osCommerce. The `tep_output_string` can be seen in Listing 4.1. The function is used to prepare strings to be printed as part of the HTML. The reason

[2] `php.net/IMPLODE`

why Eir reports so many false positives is because it currently uses the worst-case taint status for return values. If the function `tep_output_string` is called with the `$protected` parameter set to true, the function sanitizes through the `htmlspecialchars` function. If the `$protected` and `$translate` parameters are both false, the double quotes (") are replaced by the matching HTML encoding (`$quot;`). When `htmlspecialchars` is not used, the returned string is typically inserted between double quotes, which means that there is no vulnerability. However, if `tep_output_string` is used with the `$protected` parameter set to false and the returned string is inserted outside double quotes a vulnerability is most likely present. 53 of the true positives stems from calling the same two functions in os-

Listing 4.1: tep_output_string from osCommerce

```
1  function tep_output_string($string, $translate = false,
     $protected = false) {
2    if ($protected == true) {
3      return htmlspecialchars($string);
4    } else {
5      if ($translate == false) {
6        return tep_parse_input_field_data($string, array('"' => '&
           quot;'));
7      } else {
8        return tep_parse_input_field_data($string, $translate);
9      }
10   }
11 }
```

Commerce. The functions are called `tep_href_link` and `tep_catalog_href_link` where the vulnerable part of `tep_href_link` can be seen in Listing 4.2. The vulnerability is only present when the page parameter is an empty string or the connection parameter is not SSL or NONSSL. The code is vulnerable because the function parameters are printed directly in the `die` message. We have not found a way to exploit this vulnerability in osCommerce, but the reported vulnerabilities are set as true positives because they are vulnerable if the die function is called in `tep_href_link` or `tep_catalog_href_link`. In osCommerce Eir detected 134 stored XSS vulnerabilities. However, only 17 were verified. 15 of the 17 stored vulnerabilities were false positives. Most of the stored vulnerabilities reported in osCommerce are expected to be false positives, as Eir was unable to resolve the table names.

In TestLink, Eir found six true positives, where three of these were confirmed during our previous work. Two of the new true positives found were caused by user controlled `$_SERVER` variables being printed without sanitization. The last vulnerability was caused by printing every variable in the `$_POST` array without

Listing 4.2: tep_output_string from osCommerce

```
function tep_href_link($page = '', $parameters = '', $connection
  = 'NONSSL') {
  $page = tep_output_string($page);
  ⋮
  if ($page == '') {
    die('</td></tr></table></td></tr></table><br /><br /><font
      color="#ff0000"><strong>Error!</strong></font><br /><br /><
      strong>Unable to determine the page link!<br /><br />
      Function used:<br /><br />tep_href_link(\'' . $page . '\',
      \'' . $parameters . '\', \'' . $connection . '\')</strong>'
      );
  }
  ⋮
}
```

sanitization. Eir also reported false positives in TestLink. Most of the false positives were reported because of missing function specifications. Some were reported because the `urlencode`[3] function was not specified as an XSS sanitizer. The code in TestLink is safe because of the use of `urlencode`, however, it should be noted that `urlencode` should not be used as a protection against XSS vulnerabilities in general as it is only meant for URLs. Another false positive reported by Eir was because of the `json_encode`[4]. The `json_encode` function in PHP converts an associative array to a JSON formatted string. As of PHP5 the `json_encode` function also escapes dangerous characters, hence it should be regarded as a sanitization function. It is believed that these false positives would not be reported if the function specification was extended.

WordPress 3.9.2 had an XSS vulnerability that could be exploited by entering a specially formatted string into a comment section. Eir did not detect this vulnerability, but instead reported seven other vulnerabilities. All seven vulnerabilities turned out to be false positives. Three of the vulnerabilities were reported because WordPress uses custom regular expressions and string formats to sanitize the input and Eir is currently not able to handle such scenarios. Two other vulnerabilities were reported because the input was sanitized using `array_map`[5], where the sanitization function was specified as a string. The last two vulnerabilities actually included data being printed unsanitized. However, if the input was not an integer, an error

[3] `php.net/manual/en/function.urlencode.php`
[4] `php.net/manual/en/function.json-encode.php`
[5] `php.net/manual/en/function.array-map.php`

would occur and the code printing the data would never get executed.

Eir did not report any vulnerabilities in the Drupal CMS system. One false negative was present in the Drupal version analyzed. No other static tool we have tested were capable of detecting the vulnerability present in the tested version.

4.2.2 WordPress Plugins Results

Analyzing the WordPress plugins from the previous test corpus yielded better results than any of the former tools. Eir was able to find more advanced XSS vulnerabilities as well as being capable of finding several SQLi vulnerabilities. Eir detected most of the XSS vulnerabilities in the test corpus, as well as a few new vulnerabilities. The SQLi results can mainly be accredited to our framework function specifications, which are capable of modeling the WordPress database functions. Many of the false positive SQLi vulnerabilities are due to the fact that Eir does not check SQL queries for quotes. As WordPress automatically escape user input, it is safe to use user input directly in queries as long as they are within quotes. Eir was able to find four previously unknown SQLi vulnerabilities in the test corpus, but also missed one vulnerability. Overall, these results are improvements over the tools that were tested in our previous work.

The stored vulnerabilities were found in two different plugins. 18 of these were found in a single plugin. Here, the stored vulnerabilities were found by modeling the WordPress `get_options` and `update_options` in an analysis plugin. Not only did the tool find the formerly reported nine vulnerabilities from the test corpus in this plugin, but it also reported nine new vulnerabilities that were all true positives. The other plugin used the WordPress `wpdb` class to insert data into the database. In this plugin, one true positive and five false positives were found. The false positives were all due to Eir only matching storage location on the table name and not individual columns.

4.2.3 Analysis Times

When looking into the analysis times of Eir, most of the smaller and medium sized applications such as WordPress plugins and BizApp could be analyzed in a matter of minutes. Most WordPress plugins were analyzed within a few minutes, though in some plugins the analysis took close to an hour. BizApp took a little over eight minutes.

Many of the large frameworks took several days to analyze when each function call was analyzed individually. By using function summaries, the time to analyze large frameworks was reduced. Eir was able to analyze WordPress, Drupal, MODX and osCommerce in about five minutes each, whereas MyBB and TestLink took about an hour to analyze.

4.2.4 Comparing results

The results of the tests compared to the other tools can be seen in Table 4.3. Notice that when comparing to the tools from our previous work, the focus was put on reflected vulnerabilities in the comparison, as none of the static tools from our previous work found stored vulnerabilities. Therefore, the table has no stored vulnerabilities. The format in Table 4.3 is two numbers separated by a dash, where the left side is XSS and the right side is SQL (XSS-SQL). The table only includes true positives.

Table 4.3: Comparison of test results

Format: XSS-SQL	Pixy	RIPS	THAPS	w3af	Wapiti	ZAP	**Eir**
Test Application	47-0	50-23	33-0	32-3	30-17	12-0	50-20
MyBB	0-0	0-0	0-0	0-0	—	0-1	0-0
MODX	—	0-0	0-0	—	0-0	0-0	1-0
osCommerce	0-0	0-0	0-0	—	0-0	0-0	55-2
TestLink	—	3-0	0-0	—	—	0-0	2-0
WordPress	0-0	0-0	0-0	0-0	—	0-0	0-0
WP Plugins	1-0	62-0	0-0	0-0	0-0	0-0	67-7
Drupal	—	0-0	0-0	0-0	0-0	0-0	0-0
Total	**48-0**	**115-23**	**33-0**	**32-3**	**30-17**	**12-1**	**175-29**

4.3 Additional Results

As mentioned in Section 4.1, additional plugins and applications were analyzed. The results from these can be found in Table 4.4.

4.3.1 BizApp Results

Looking into the results of BizApp, Eir reported 497 vulnerabilities. While this result seems overwhelming, many of the reports are sensible and understandable. Most of the XSS vulnerabilities found are due to user input from `$_GET` being used directly in an `echo` command. This happens often in BizApp when constructing the HTML of a page. Most of the SQL vulnerabilities found are of the same reason. Input from `$_GET` is concatenated directly to SQL queries without being sanitized. It should be noted that some of the true positive XSS vulnerabilities found are hard, if not impossible, to exploit in the current system because of conflicting vulnerabilities, lack of error handling and crashes in the system when the system receives invalid input. The reason these are still set as true positives is because it is clear from the

Table 4.4: Results of Eir scans

Eir	SQLi			sXSS			rXSS		
	TP	FP	UN	TP	FP	UN	TP	FP	UN
BizApp	15	3	57	15	0	202	15	0	148
Ninja Forms 2.9.17	0	0	0	0	0	0	5	2	0
Fast Secure CF 4.0.37	0	0	0	0	0	0	0	5	0
WP Super Cache 1.4.4	0	0	0	2	0	0	2	1	0
Cart66 Lite 4.1.5	0	2	0	0	0	0	1	0	2
Contact Form DB 2.8.38	0	0	0	0	0	0	3	1	0
Google Doc Emb 2.5.19	1	3	0	0	0	0	2	3	0
WP Construction Mode 4.1.5	0	0	0	0	0	0	0	0	0
Like Dislike Counter 1.3.0	0	0	0	0	0	0	0	0	0
Spider Facebook 1.0.12	0	0	0	0	0	0	0	0	0
Photo Gallery 1.2.31	0	0	0	0	0	0	5	0	0
Total	16	8	57	17	0	202	33	12	150

UN = Unconfirmed vulnerabilities.

code that if other vulnerabilities and crashes were patched, the vulnerabilities would be exploitable.

The stored vulnerabilities found were present in simple forms where it was possible to save information on tasks and customers. Here is was possible to insert XSS and SQL payloads that could be used later to attack both other users and the database. As an example, the stored SQLi vulnerabilities allow an attacker to store an SQLi payload in the database, that would be activated when a user visited certain pages in the web application.

All of the confirmed false positives were false positives because of integer and datetime checks. These checks were performed in if-statements, where the code would always enter if the input was not a proper format. If an improper format was detected, it would be set to a default value. However, due to the nature of our taint analysis, Eir looks at both the path where the code enters the if-statement and the case where it does not, and thus reports a vulnerability when the user input is printed in an `echo`.

Note that as there were 497 vulnerabilities reported, not all vulnerabilities were confirmed. Many were inspected by looking into the code, but only some were tested in the running application. Certain parts of the application sanitizes by validating the input, but this is rare and most of the reported vulnerabilities are likely true positives. A total of 15 vulnerabilities of each type were confirmed.

4.3.2 WordPress Plugins

Ninja Forms[6] is a WordPress plugin from the popular page with over 200.000 active installs. It gives a WordPress site a lot of customization to contact forms. The plugin thus also has a lot of input fields and settings, which can naturally be vulnerable if not handled properly. Ninja Forms was analyzed using Eir, where it was able to find 5 new confirmed XSS vulnerabilities. The vulnerabilities were all simple to exploit, but required that the victim was logged in as an administrative user on the site.

Fast Secure Contact Form[7] provides contact forms meant to deter spam bots amongst other things. The plugin has over 400.000 active installs, but no true positive vulnerabilities were detected. The false positives present were detected because the plugin uses regular expressions to validate email addresses. It also uses the PHP `sprintf` method to format the strings that are echo'ed to the page. In these, the plugin defines that the tainted input has to be printed as decimals, thus automatically sanitizing the input. A simplified example of this code can be found in Listing 4.3. Here, `$this_form` is tainted, but is inserted into the string placeholder `%d`, which casts to a decimal.

Listing 4.3: Safe echo of tainted $this_form

```
1 echo sprintf( __( 'Form %d ', 'si-contact-form' ), $this_form );
```

WP Super Cache[8] is a plugin that was featured on the WordPress plugin pages with over 1.000.000 active installs. This popular plugin is meant to speed up a WordPress site. In this plugin, Eir reported five vulnerabilities. Two reflected and two stored XSS vulnerabilities were confirmed as true positives.

The last seven plugins are all WordPress plugins from our previous work that have received patches. Two of the true positive vulnerabilities in Google Doc Embedder are marked as true positives even though they cannot be exploited with the current version of the plugin. The reason for this is that the code is in fact vulnerable, but the code is unreachable. If the code becomes reachable in a future version of the plugin, the vulnerability can be exploited. Many of the false positive XSS vulnerabilities were reported because the application changed the `content-type` header, which makes it impossible to execute scripts in the browser on that page. The false positive SQLi vulnerabilities were primarily due to quotes used in the queries. In Cart66 Lite, two of the vulnerabilities cannot be verified. One of the vulnerabilities requires the pro version of the plugin to reach the vulnerable code and the other

[6] https://wordpress.org/plugins/ninja-forms/
[7] https://wordpress.org/plugins/si-contact-form/
[8] https://wordpress.org/plugins/wp-super-cache/

requires a PayPal token to reach the vulnerable code. However, both vulnerabilities print out data from the `$_POST` array without sanitization, which means both of the vulnerabilities are most likely true positives. The Contact Form 7 DB plugin was analyzed and four vulnerabilities were reported. One false positive prints a variable from the `$_REQUEST` array, which holds a username. The username is printed directly without sanitization. However, the username variable cannot contain HTML tags or JavaScript and is therefore deemed a false positive. Two true positives were reported by Eir where a string is retrieved from the `$_REQUEST` array and printed two times in a HTML form without sanitization, thus two vulnerabilities exists.

CHAPTER 5
EVALUATION

This chapter discusses and evaluates the developed tool in terms of the results and findings made throughout the report. The chapter is structured with an evaluation in Section 5.1. Furthermore, Section 5.2 discusses possible areas of interest, if the work is to be continued.

5.1 Evaluation

Eir was developed with extensibility and reusability in mind, which has already been useful. As an example, Eir was used to count `goto` statements in tens of thousands of WordPress plugins as stated in Section 3.2.

Another example showing Eir's extensibility is another project that already uses Eir. The project expands on the CFG creator by creating a super CFG, instead of creating a lot of small CFGs[13].

During the project, a test suite has been maintained and as stated in Section 1.4, the test suite is there to ensure correct code. The test suite has served as a safety precaution for the tool during development. A couple of times the test suite has helped to find bugs in the code early, allowing them to be fixed quickly. Tests have also been helpful when it comes to documenting not implemented features and bugs. A failing test case would be implemented for the unsupported behavior in order to show what needed to be done to support it. This has helped the project as specific scenarios could be immediately tested when implemented.

The tool was also created to detect stored vulnerabilities. Using simple detection techniques to detect when data is inserted into a storage location and taken back out again, Eir is able to find a decent amount of stored vulnerabilities. With further improvements in detecting storage locations, the results will become more precise

and more vulnerabilities will potentially be found.

Eir also implements the support for analyzing framework extensions. This has been demonstrated by the WordPress plugin created for Eir. This module makes use of the extensibility of Eir and is created as a plugin for Eir to detect WordPress functions and methods. This is to make sure that the analysis reacts correctly upon them without having to scan the WordPress core. Eir plugins are able to make use of all the principles and tools of the Eir analysis. It was therefore possible to find stored vulnerabilities in WordPress plugins, where data was saved using WordPress native functions for inserting and extracting data from the database.

Eir showed good results in both vulnerabilities reported and speed. Some of the fastest and best results were found in the user created WordPress plugins and projects with less focus on security.

Initially, Eir proved to be slow when analyzing larger frameworks, such as WordPress, Drupal and TestLink. The problem was that whenever the analysis encounters a function call, the analysis will look up the function called and perform a full analysis of the function. Large frameworks typically have many abstraction layers, which are used many places throughout the framework. Therefore, Eir includes the option to use function summaries in the analysis so that if a function is called multiple times with arguments having the same taint status, the output taint status of the function is returned immediately. By analyzing large frameworks with these summaries the time taken to analyze a framework was reduced drastically. For example, an attempt to analyze TestLink without summaries was canceled after four days of analyzing, whereas the analysis with function summaries took about one hour. Even though using function summaries makes the analysis less precise, it was necessary with the larger frameworks.

On top of scanning the test corpus from our previous work, Eir was also used to analyze several new WordPress plugins and updated versions of previously analyzed plugins, in which Eir reported several true positive vulnerabilities.

5.1.1 False Positives

Eir reported false positive SQLi vulnerabilities. The reason is that Eir currently does not analyze strings for quotes when an SQL statement is formed. Quotes can help protect against SQLi, which is why Eir is likely to report a vulnerability when there is none. The number of false positives reported because of this can be reduced by analyzing quotes. It has not been implemented in Eir as the focus has been on other problems. However, it has been shown in other tools, that it is possible to map quotes in SQL statements precisely[7].

Another source of false positives is due to missing function specifications, which should be easy to resolve, as the functions should just be specified in the JSON files. However, some of the sanitization functions cannot be handled correctly in

the current implementation of Eir as they are parameter dependent. Therefore, the analysis should consider the parameter specifications in the JSON files.

5.1.2 Problematic Code

During development and testing of Eir, cases that are hard to analyze using a static analysis tool were found. Cases that have caused false positives have already been discussed in Section 4.2. Some cases have not been causing false positives, but are problematic for a static analysis tool. Throughout this section, some problematic code examples are explained.

By analyzing the Drupal CMS, code was found that instantiates objects dynamically. The code shown in Listing 5.1 uses an associative array (called `batch_set`). This associative array holds a `queue` key. The value of the `queue` key is an array which holds the keys `name` and `class`. The code then instantiates objects based on the values from the `name` and `class` keys in the array. This is problematic for a static analysis as the analysis has to know the values of `$name` and `$class` to precisely analyze the code. The array may be filled with data that is only accessible at runtime, which means that a static analysis has no way of analyzing which objects are instantiated. Furthermore, as the instantiation of objects is based on string values, the analysis tool has to be capable of mapping strings to class names and hereafter use this information during the analysis.

Listing 5.1: Dynamic object instantiation in Drupal.

```
1 if (isset($batch_set['queue'])) {
2   $name = $batch_set['queue']['name'];
3   $class = $batch_set['queue']['class'];
4
5   if (!isset($queues[$class][$name])) {
6     $queues[$class][$name] = new $class($name);
7   }
8   return $queues[$class][$name];
9 }
```

Code was also found that calls functions based on function names given as a string. Analyzing dynamic function calls is problematic in the same way dynamic object instantiations are. Code to show an example of this is given in Listing 5.2

Listing 5.2: Dynamic function call in Drupal.

```
1 if (isset($elements['#pre_render'])) {
2   foreach ($elements['#pre_render'] as $function) {
3     if (function_exists($function)) {
4       $elements = $function($elements);
5     }
6   }
7 }
```

5.2 Future Work

Value Approximation Knowing/approximating what possible values a variable could have at any given point in a program execution could be very useful in the analysis. It would allow for better analysis of possible execution paths, as well as better precision when it comes to determining whether a variable has been sanitized or not. Value analysis has been explored in Christensen et al. [3], where flow analysis is used to generate regular languages that describe possible outcomes of string expressions in Java programs. While the analysis used in [3] can not necessarily be applied directly to our taint analysis, the ideas and techniques can be used as inspiration for improving the analysis of PHP applications.

File Inclusion Dynamic inclusion of files in PHP is hard to reason about statically and is generally not handled very well in our prototype. More advanced inclusion resolving is required to properly analyze many PHP applications. Static inclusion resolving in PHP applications has previously been explored in Hills et al. [6] and the techniques used should be applicable to our analysis.

False Positive Reduction An analysis may report vulnerabilities that are not actual vulnerabilities, which may confuse the user of an analysis tool. The number of false positives should be reduced if possible, such that as many as possible of the reported vulnerabilities are true positives. This is both an improvement to the analysis tool itself, but will also help the user of the analysis to correct more vulnerabilities such that the analyzed web application may become more secure.

Improved Stored Vulnerability Handling When creating our prototype, the SQL handling created for stored vulnerability analysis was simplistic and made as a proof of concept. It is possible to further develop this to detect all possible SQL commands for changing and fetching data from a database. Furthermore, it would be beneficial to do more work on detecting exact storage locations.

Currently the prototype only detects the table name. Further improvement on this area could involve tracking columns in SQL in order to reduce false positives.

The current implementation only checks for stored vulnerabilities in databases. However, databases are not the only stored locations that can produce security issues. Files may also store malicious data therefore support for finding stored vulnerabilities in files is to be implemented. Sessions are typically short-lived, but data can be stored throughout a session. Support for session handling in regards to finding stored vulnerabilities should also be implemented.

Context Aware Analysis Currently the implementation only parses and analyzes PHP code. However, as shown in the Test Application it is not always sufficient. There exists cases where the analysis needs to be context-aware in regards to HTML. This is because some vulnerabilities may depend on specific HTML tags to be used. The tool could be expanded with knowledge about HTML context so that vulnerability detection will be more precise in regards to detecting XSS vulnerabilities. Furthermore, as can be seen in the WordPress plugin results, analyzing whether quotes exist in an SQL query is important in order to reduce false positives. Since Eir already has the capability of tracking various degrees of taint, the analysis would need to analyze the SQL queries in an application and detect whether quotes are present.

Regular Expressions Regular expressions can be used for sanitizing, however, regular expressions are defined by the developer. It is suggested to whitelist regular expression patterns that are known to sanitize data, such that the analysis can make use of this knowledge to make the analysis more precise. The JSON specification developed could capture regular expression matching, as functions such as `preg_replace`[1] and `preg_match`[2] need the regular expression as input, and thus could be specified in the values array in the JSON specification.

Since it is not reasonable to whitelist all safe regular expressions, a better approach to handling them would be to implement a proper regular expression analysis. Such an analysis would detect regular expressions in the code and analyze them in order to figure out if the regular expression changes the taint status of a variable.

Other Analysis In the current state, the tool only employs taint analysis in regards to finding SQLi and XSS vulnerabilities. The tool could also be extended with other analyses such as the Reaching Definition analysis. The tool could be

[1] `php.net/manual/en/function.preg-replace.php`
[2] `php.net/manual/en/function.preg-match.php`

extended to find other types of vulnerabilities, such as file disclosure, directory traversal or command injection vulnerabilities.

Function Summaries The currently implemented function summaries in Eir are simplistic. When selecting a function summary, only the parameters' taint status is used. This approach ignores changes to global variables between two function calls, which can result in missed vulnerabilities. It would be beneficial to implement summaries that represent the context of a function call more precisely, as well as storing more relevant information in the summary.

CHAPTER 6

CONCLUSION

This report has documented the development of a static analysis tool for finding web application vulnerabilities in PHP applications. The tool, called Eir, is capable of finding reflected and stored Cross-site scripting (XSS) and SQL injection (SQLi) vulnerabilities. The work in this report is a continuation of previous work[17], where six existing web application analysis tools, which were all capable of finding vulnerabilities in web applications, were tested and evaluated. Our previous work focused on XSS and SQLi vulnerabilities as they are some of the most common web application vulnerabilities and therefore the work in this report has also focused on these vulnerability types.

After evaluating the existing tools, it was decided that Eir needed to be implemented with extensibility and code quality in mind. Eir employs an architecture that is built up in modules, such that individual parts of the analysis tool can be used and extended individually. Eir also has a test suite, which was not common for existing tools. The test suite is meant to work as a safety precaution when making changes in the implementation but also works as documentation for the functionality. The test suite should also help others learn and extend Eir.

We also found a need for a tool that supports stored vulnerability detection, as none of the static tools evaluated previously supports this. Eir implements an approach where potentially stored vulnerabilities are found across multiple data flow paths, which is one of the main contributions of Eir. The tool is currently capable of tracking database tables from query strings when finding stored vulnerabilities.

Eir also uses a JSON formatted function specification where other tools use whitelisting of function names. The reason Eir does this is that the analysis can be made parameter dependent in regards to function specifications. The JSON format can be extended with new PHP functions later on. Framework functions can also

be specified in the JSON format, which will make information about framework functions available to the analysis, which is useful when analyzing extensions.

Furthermore, framework models were implemented in order to support analyzing framework extensions more conveniently. This allows Eir to quickly analyze extensions, as the framework itself does not need to be analyzed. It is shown how this is supported through the plugin architecture that is implemented in Eir. This plugin architecture allows users of Eir to write their own plugins for the analysis to allow for customization of the analysis, as it is hard to build a tool that suits all needs. A plugin was created to show how the plugin architecture works and is a plugin for analyzing WordPress extensions. The created plugin handles special WordPress functionality and functions that require more than just a JSON formatted function specification. This has helped find vulnerabilities in WordPress plugins.

Eir was used to analyze the same test corpus as in our previous work. In addition to this Eir was used to find 21 new confirmed vulnerabilities in new popular WordPress plugins. Not only did the tool find 18 new XSS and one new SQLi, but it was also able to find two new stored vulnerabilities. The two stored vulnerabilities were detected in a WordPress plugin with over 1.000.000 million active installs.

Eir was also used to analyze a local business' application. Here, Eir reported 497 vulnerabilities, where 15 stored vulnerabilities, 15 reflected SQLi and 15 reflected XSS were actually confirmed. The rest of the vulnerabilities were left unconfirmed. Eir also reported false positives, most of which were caused by types being verified as datetimes and numbers.

BIBLIOGRAPHY

[1] Robert Auger. Cross site scripting. `http://projects.webappsec.org/w/page/13246920/Cross%20Site%20Scripting`, 2014. Retrieved 2014-11-07.

[2] Checkmarx Ltd. The Security State of Wordpress' Top 50 Plugins. `https://www.checkmarx.com/wp-content/uploads/2013/06/The-Security-State-of-WordPress-Top-50-Plugins3.pdf`, June 2013. Retrieved: 2015-03-18.

[3] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. *Precise analysis of string expressions*. Springer, 2003.

[4] Johannes Dahse. RIPS - A static source code analyser for vulnerabilities in PHP scripts. `http://www.php-security.org/downloads/rips.pdf`, 2011. Retrieved 2014-11-27.

[5] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of PHP feature usage: a static analysis perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 325–335. ACM, 2013.

[6] Mark Hills, Paul Klint, and Jurgen J Vinju. Static, lightweight includes resolution for php. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 503–514. ACM, 2014.

[7] Torben Jensen, Heine Pedersen, Mads Chr. Olesen, and René Rydhof Hansen. THAPS: Automated vulnerability scanning of PHP applications. In *Secure IT Systems*, pages 31–46. Springer, 2012.

[8] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (technical report). *Secure Systems Lab, Vienna University of Technology*, 2006.

[9] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[10] Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis - Theory and Practice*. Springer-Verlag Berlin Heidelberg, 1999.

[11] National Vulnerability Database. XSS Vulnerabilities - 2014. `https://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&cves=on&cwe_id=CWE-79&mod_date_start_month=0&mod_date_start_year=2014&mod_date_end_month=11&mod_date_end_year=2014&cve_id=`, 2015. Retrieved: 2015-03-17.

[12] National Vulnerability Database. SQL Injection Vulnerabilities - 2014. `https://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&cves=on&cwe_id=CWE-89&mod_date_start_month=0&mod_date_start_year=2014&mod_date_end_month=11&mod_date_end_year=2014&cve_id=`, 2015. Retrieved: 2015-03-17.

[13] Jens Thomas Vejlby Nielsen. Detecting incorrect wordpress plugin function usage, 2015.

[14] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. CRC Press, 2009.

[15] Nikita Popov (nikic). Php-parser. `https://github.com/nikic/PHP-Parser`, April 2015. Retrieved: 2015-04-01.

[16] OWASP. Cross-site scripting (XSS). `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`, 2014. Retrieved 2014-11-07.

[17] Mikkel-Alexander Vej, Morten Nørtoft, and Kenneth Michael Jepsen. Survey of Web Security Analysis Tools. Technical report, 2014.

[18] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, volume 6, pages 179–192, 2006.

APPENDIX A

WORDPRESS PLUGINS IN TEST CORPUS

Table A.1: Test corpus

	Version	SQL Injections	Stored XSS	Reflected XSS	Total
WP Construction Mode	1.8	0	9	1	10
Photo Gallery	1.1.30	0	0	6	6
Contact Form DB	2.8.13	0	0	39	39
Google Doc Embedder	1.5.14	1	0	2	3
Cart66 Lite WordPress eCommerce	1.5.1.17	1	0	2	3
WordPress Like Dislike Counter	1.2.3	1	0	0	1
Wordpress Spider Facebook	1.0.8	1	0	14	15
Simple Visitor Stat	1.0	0	1	1	2
Total	-	4	10	65	79

APPENDIX B

FILE WRITER

This is an example of a simple plugin that can be used with Eir while analyzing. This plugin write basic vulnerability information to a file. If present, this plugin is included in the analysis and used along with other vulnerability reporters whenever a vulnerability is found.

Listing B.1: File writer plugin

```
1  [Export(typeof(IVulnerabilityReporter))]
2  public sealed class FileVulnerabilityReporter :
     IVulnerabilityReporter
3  {
4    private readonly string _vulnerabilityFile;
5    private int vulnCounter = 1;
6    public FileVulnerabilityReporter()
7    {
8      this._vulnerabilityFile = "scan-report.txt";
9    }
10    ∇
```

Listing B.2: File writer plugin continued

```
10      ∇
11    public void ReportVulnerability(IVulnerabilityInfo
        vulnerabilityInfo)
12    {
13      WriteBeginVulnerability();
14      WriteInfoLine("Message: " + vulnerabilityInfo.Message);
15      WriteInfoLine("Include stack:" + String.Join(" -> ",
          vulnerabilityInfo.IncludeStack));
16      WriteInfo("Call stack: " + String.Join(" -> ",
          vulnerabilityInfo.CallStack.Select(c => c.Name)));
17      WriteEndVulnerability();
18    }
19
20    public void ReportStoredVulnerability(IVulnerabilityInfo[]
        vulnerabilityPathInfos)
21    {
22      WriteBeginVulnerability();
23      foreach (var pathInfo in vulnerabilityPathInfos)
24      {
25        WriteInfoLine(">> Taint Path: ");
26        WriteInfoLine(pathInfo.Message);
27        WriteInfoLine(String.Join("->", pathInfo.IncludeStack));
28        WriteInfoLine("Callstack: " + String.Join(" -> ", pathInfo.
            CallStack.Select(c => c.Name)));
29      }
30      WriteEndVulnerability();
31    }
32
33    private void WriteBeginVulnerability()
34    {
35      File.AppendAllLines(_vulnerabilityFile, new[] { "|> " +
          vulnCounter++ });
36    }
37    private void WriteInfo(string info)
38    {
39      File.AppendAllText(_vulnerabilityFile, info);
40    }
41
42    private void WriteInfoLine(string info)
43    {
44      WriteInfo(info);
45      WriteInfo(Environment.NewLine);
46    }
47    private void WriteEndVulnerability()
48    {
49      File.AppendAllLines(_vulnerabilityFile, new[] { "", "<|" });
50    }
51  }
```

APPENDIX C

EIR SOURCE CODE INFORMATION

The source code of Eir can be found on GitHub with the following link: `https://github.com/Chikila/Eir`.

The application at the time of hand-in is marked with Release Tag 0.0.1, corresponding to Git commit `3825aea9d42325c6cc92f923a6a9afe70918b70b`.