Wright State University
CORE Scholar

Browse all Theses and Dissertations

Theses and Dissertations

2016

Detecting PHP-based Cross-Site Scripting Vulnerabilities Using Static Program Analysis

Steven M. Kelbley
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
Kelbley, Steven M., "Detecting PHP-based Cross-Site Scripting Vulnerabilities Using Static Program Analysis" (2016). *Browse all Theses and Dissertations*. 1655.
https://corescholar.libraries.wright.edu/etd_all/1655

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.

Detecting PHP-based Cross-Site Scripting Vulnerabilities Using Static Program Analysis

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Cyber Security

by

Steven M. Kelbley
B.S. Biological Sciences, Wright State University, 2012

2016
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

December 1, 2016

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Steven M. Kelbley ENTITLED Detecting PHP-based Cross-Site Scripting Vulnerabilities Using Static Program Analysis BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

____
Junjie Zhang, Ph.D.
Thesis Director

____
Mateen Rizki, Ph.D.
Chair, Department of Computer
Science and Engineering

Committee on
Final Examination

____
Junjie Zhang, Ph.D.

____
Krishnaprasad Thirunarayan, Ph.D.

____
Adam Bryant, Ph.D.

____
Robert E. W. Fyffe, Ph.D.
Vice President for Research and
Dean of the Graduate School

ABSTRACT

Kelbley, Steven. M.S. Cyber Security, Department of Computer Science, Wright State University, 2016. *Detecting PHP-based Cross-Site Scripting Vulnerabilities Using Static Program Analysis.*

With the widespread adoption of dynamic web applications in recent years, a number of threats to the security of these applications have emerged as significant challenges for application developers. The security of developed applications has become a higher priority for both developers and their employers as cyber attacks become increasingly more prevalent and damaging.

Some of the most used web application frameworks are written in PHP and have become major targets due to the large number of servers running these applications worldwide. A number of tools exist to evaluate PHP code for issues, however most of these applications are not targeted at vulnerability detection. At the same time, Cross-Site Scripting (XSS) vulnerabilities continue to be identified in existing software threatening the security of client data. Providing tools to software developers which can identify these XSS vulnerabilities in code during the development process could reduce the number of vulnerabilities that make it into production code and thus threaten users.

This thesis proposes a solution for the problem of identifying non-persistent XSS vulnerabilities in PHP code by demonstrating a system which is capable of finding these vulnerable code paths. This is achieved through the use of static taint analysis, whereby a number of known sources of untrusted data are defined, along with several sensitive sinks which may present a vulnerability if untrusted data is used at these locations. Any data acquired from these taint sources and subsequent propagation of the data is tracked.

Code analysis is performed on an Abstract Syntax Tree (AST), an intermediate representation which permits conversion to and from source code. This allows individual line numbers to be tracked for the purpose of clearly displaying taint flow to the user allowing them to visualize how the information flow could result in an unsafe condition and take

appropriate action to remedy the vulnerability.

This program is capable of analyzing non-object-oriented PHP code and supports most of the common language constructs. Initial testing has shown the program to be highly successful at identifying non-persistent XSS attacks in the supported subset of the PHP language, with future development efforts targeting expanded support for more elements of the language including object-oriented programming.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Background . . . 2
  - 1.2.1 Static Analysis . . . 2
  - 1.2.2 Dynamic Analysis . . . 3
  - 1.2.3 Taint Analysis . . . 3
  - 1.2.4 Cross-Site Scripting (XSS) . . . 4
- 1.3 Goals . . . 5
- 1.4 Organization . . . 6

2 Related Work **7**

3 Design **11**
- 3.1 Problem Formulation . . . 11
- 3.2 Intermediate Representation . . . 11
- 3.3 Syntax Rules . . . 12
- 3.4 Taint Flow Analysis . . . 13
  - 3.4.1 Sources . . . 13
  - 3.4.2 Sinks . . . 14
  - 3.4.3 Sanitizers . . . 14
  - 3.4.4 Approximation . . . 15
- 3.5 Evaluating Nodes . . . 20

4 Implementation **24**
- 4.1 Variable Taint . . . 24
- 4.2 Function Definitions . . . 27
- 4.3 Taint Propagation . . . 27
  - 4.3.1 Taint Propagation Functions . . . 28
  - 4.3.2 Hard-coded Expressions . . . 29
  - 4.3.3 Variable Assignments . . . 29
  - 4.3.4 Binary Operations . . . 30
  - 4.3.5 Loops . . . 31

4.3.6 Conditionals . . . 34
4.3.7 Functions . . . 38
4.3.8 Sinks . . . 40

5 Evaluation **42**
5.1 Samples . . . 42
5.2 Effectiveness . . . 45
5.3 Running Time . . . 46

6 Discussion **47**

7 Conclusion **50**

Bibliography **50**

A Appendix A **54**

B Appendix B **76**

List of Figures

5.1 Effect of AST Node Count on Running Time 46

List of Tables

3.1 PHP Syntax 12
3.2 Expression Syntax Rules 21
3.3 Statement Syntax Rules 21

4.1 Variable Taint Data Structure 26

5.1 Taint Flow 46

Listings

1.1 Example XSS Vulnerability . . . 5
3.1 Taint Overapproximation . . . 16
3.2 Branch Execution with Default . . . 17
3.3 Example Code . . . 19
3.4 Example AST . . . 19
4.1 Variable Taint Data Structure . . . 25
4.2 Sample Taint Generation . . . 25
4.3 Assignment Operation Example . . . 29
4.4 Assignment Operation Evaluation . . . 30
4.5 Binary Operation Example . . . 31
4.6 Binary Operation Evaluation . . . 31
4.7 While Loop Example . . . 31
4.8 While Loop Evaluation . . . 32
4.9 For Loop Example . . . 32
4.10 For Loop Evaluation . . . 33
4.11 Foreach Loop Example . . . 34
4.12 Foreach Loop Evaluation . . . 34
4.13 If-Else Conditional Example . . . 35
4.14 If-Else Conditional Evaluation . . . 36
4.15 Switch Example . . . 37
4.16 Switch Evaluation . . . 37
4.17 Function Example . . . 39
4.18 Function Evaluation . . . 39
4.19 Sinks Example . . . 40
4.20 Sinks Evaluation . . . 40
5.1 Example 1 Program Output . . . 43
5.2 Example 2 . . . 44
6.1 PHP Array Gaps . . . 48
A.1 Program Code . . . 54
B.1 Example 1 . . . 76

Acknowledgment

I would like to take this opportunity to extend my thanks to my advisor, Dr. Junjie Zhang, for his guidance and mentoring while I completed this thesis. His experience and expertise has been invaluable in assisting my work. He has greatly contributed to my education at Wright State and I appreciate his support. I would also like to thank Dr. Krishnaprasad Thirunarayan and Dr. Adam Bryant for volunteering their time to serve on my thesis committee and sincerely appreciate their input and expertise in evaluating this work. I am thankful for the time everyone has invested to assist me in this final chapter of my graduate education. I would like to thank my family for their support during my academic career and helping provide me with the opportunity to study at Wright State University. Finally, I would like to thank my wife for supporting me while I pursued further education and for tirelessly believing in me.

Abbreviations

AST — Abstract Syntax Tree

CFG — Control Flow Graph

CMS — Content Management System

DBMS — Database Management System

HTML — Hypertext Markup Language

IR — Intermediate Representation

IT — Information Technology

OOP — Object Oriented Programming

PoC — Proof-of-Concept

PHP — PHP Hypertext Preprocessor

SQL — Structured Query Language

SQLI — SQL Injection

XSS — Cross-Site Scripting

Introduction

In this thesis, a solution is proposed that can analyze non-object-oriented PHP code for reflected Cross-Site Scripting (XSS) vulnerabilities. Specifically, we will discuss the implications of XSS vulnerabilities in PHP code and ways in which malicious actors can inject unsafe scripts into pages as a result of poor coding practices. We will demonstrate the use of an abstract syntax tree (AST) in our analysis of code to allow feedback to be displayed in a human-readable manner, allowing programmers to rapidly identify the code which permits the XSS attack so that it can be corrected.

1.1 Motivation

An ever-increasing number of critical applications are moving to web-based systems as developers attempt to create applications that can be utilized on a wide range of devices via a web browser. As a consequence, this broad adoption has brought increased attention to the vulnerabilities in these programs that can be exploited by malicious users[1]. The OWASP Foundation [2] analyzed over 500,000 vulnerabilities in 2013 to discern what types were most commonly exploited, and determined that SQL injection (SQLI), issues with Authentication and Session Management and Cross-Site Scripting (XSS) were the top three threats to web applications. Scott and Sharp[3] contend that web application vulnerabilities are inevitable regardless of the language used and the implementation of the various services on which the application relies.

Among the most popular languages for web applications is PHP, which was used on

over 240 million sites as of January 2013 [4]. Many high-traffic web applications and web development frameworks (and their plugin modules) are written in PHP, including Wordpress[1], Drupal[2], Laravel[3] and CodeIgniter[4]. Wordpress alone accounts for over 73 million sites on the internet with over 300 million unique visitors each month[5]. However, the widespread use of frameworks like Wordpress has attracted malicious individuals due to the uniformity of these installations and the inevitable delay between security patches being committed and system administrators deploying the patches.

With such a high number of websites running on programs and frameworks written in PHP, static analysis of PHP code can prove to be a valuable tool to the developers of these software programs. The rapid identification of vulnerabilities in code is paramount to the security of these sites and presents a major challenge for developers and system administrators. In the current cybersecurity climate, it is critical to ensure web applications can securely process and store client data.

1.2 Background

Several analytical techniques are utilized in this work. Below we provide some background information on these approaches.

1.2.1 Static Analysis

Static analysis is the term that describes the examination of program code without executing the program[6]. This approach can evaluate all of the code in a program for possible vulnerabilities and provides a thorough approach for identifying issues that human code inspection may fail to detect. As the analysis is not dependent on a single execution

[1] https://wordpress.com/
[2] https://www.drupal.org/
[3] https://laravel.com/
[4] https://www.codeigniter.com/

path, it is possible to evaluate all branches individually. Static analysis serves as a jumping-off point for using further analytical methods to solve the problem at hand.

1.2.2 Dynamic Analysis

Dynamic analysis is an analytical technique in which a program is run with test inputs and the execution is monitored for interesting (or undesirable) output or behavior[7]. Since the outputs are dependent on the inputs, identification of vulnerabilities is dependent on generating test inputs that reach the code path on which the vulnerability lies; therefore, the careful creation of test inputs is required to ensure sufficient code coverage. Additionally, any instrumentation code added for the purpose of performing dynamic analysis could potentially cause problems in programs where time-sensitive operations occur. For these reasons, static analysis was chosen as we seek to evaluate as much of the code as possible for vulnerabilities.

1.2.3 Taint Analysis

In this thesis, we use static analysis to evaluate program flow paths in source code. The goal of this analysis is to identify points where execution could potentially result in tainted input (data retrieved from an untrusted source) being displayed within the client browser[8]. There are multiple representations in which the program could be analyzed: source code, abstract syntax trees and various intermediate representations all provide valuable information about how the program functions.

Parsing source code allows easy identification of culprit lines of code, however this requires a significant amount of work to handle the specific syntax of the PHP language. Intermediate representations (IR) permit simple flow analysis and are commonly used by compilers for this purpose, but converting the IR back to source code may result in different syntax than was used in the original program and could make it difficult for users to find errors in the original source code. Abstract syntax trees represent the source code as a tree

data structure which abstracts away specific syntax elements of the code, such as grouping of parentheses, and allows for easy traversal of the tree (and thus the source code). Additionally, the AST is readily converted back to the exact original source code, making it easy to point out specific line numbers for vulnerabilities. As such, we elected to use an AST-based approach.

Taint flow analysis allows us to track the flow of this tainted data through the AST (and thus the program), identifying where it is generated, where it is transferred to other data structures, and where it may be used in sensitive operations such as outputting to the screen. Due to the nature of information flow analysis and the use of a single trace of the execution, the results are an approximation of the data flow. However, this does not diminish the value of the data output by the program, as it would still be capable of clearly identifying sensitive sinks for data and which taint sources generated the untrusted data in question, allowing users to see the problem and determine how best to fix the vulnerability.

1.2.4 Cross-Site Scripting (XSS)

Cross-site scripting vulnerabilities have been a major threat to web applications for a number of years, ranking at second and third on the list of most critical vulnerabilities in 2010 and 2013, respectively[9]. These vulnerabilities pose a risk to clients, as they can trick users into executing untrusted code by luring them into clicking a link that contains parameters which cause code to be printed back to the page displayed on the client. This code can then cause adverse effects on the client ranging from minor annoyance to major security vulnerability, depending on the desired effect of the exploit's author(s)[10].

XSS attacks provide an avenue by which attackers can bypass the same-origin policy implemented by web browsers which prevents scripts loaded from one origin (a combination of hostname, port and Uniform Resource Identifier) from accessing data used by scripts from another origin[11]. An XSS attack would appear to be coming from the same origin as the site with the vulnerability, and would therefore be permitted to access cookies

and other data stored by the target site.

A simple example of vulnerable code is shown below in Listing 1.1. Two values are retrieved from the PHP built-in array $_POST at the indices 'name' and 'zip_code'. If the user has supplied a ZIP code, then they are thanked by name; otherwise, they are informed that the ZIP code field is required.

Listing 1.1: Example XSS Vulnerability

```
1  $name = $_POST['name'];
2  $zip = $_POST['zip_code'];
3
4  $response = '';
5  if($zip){
6      $response = "Thank you for your submission, " . $name . "\n";
7  } else {
8      $response = "Your ZIP code is required\n";
9  }
10
11 echo $response;
```

Preventing XSS attacks entails sanitizing data before it is returned to the client browser for output. In the case of the example above, if the '$response' variable had been provided to a sanitization function like 'htmlspecialchars', any code that was passed in via the 'name' parameter would be displayed on the page for the user in HTML-escaped form; the <script> tag would be converted to <script> and the script would no longer be valid Javascript capable of being executed.

1.3 Goals

In this thesis, we designed and implemented a system which is capable of finding reflected XSS vulnerabilities by performing static analysis on PHP code using taint analysis to identify potentially unsafe operations on input data. As much of the PHP language as possible should be handled to create an accurate representation of the data flow of the target

program, excluding operations that would not play a direct role in reflected XSS attacks (e.g. file operations or database queries). Excluding OOP code (for reasons mentioned previously) and these ‘unrelated’ operations, a high percentage of commonly-used elements of the PHP language are supported in this work.

The system should be capable of reliably detecting operations which could result in untrusted input being displayed on a web page, a condition which allows XSS to occur. We specifically target reflected (non-persistent) XSS, in which data is passed into the page directly from the client. The intermediate representation chosen should allow the ability to be converted back into source code for display to the user of the software; this capability allows us to precisely show from where the untrusted input originated, all operations in which it was propagated, and finally where the untrusted input was used in an operation which resulted in the content being displayed in the web page.

The supported set of syntax elements of PHP, and thus the scope of this work, was chosen based on the most commonly-used elements of the language. Standard conditional and loop constructs are supported, and most use-cases of language elements were handled. Support for object-oriented code was omitted as it presented a separate non-trivial problem and was left for later work.

1.4 Organization

Chapter 1 describes the motivation for this work, background on used techniques, the goals of the project and the procedure for validating results. Chapter 2 provides an overview of related work in this research area and details approaches taken by other authors towards solving this problem. Chapter 3 details the approach used in this work and the techniques employed for vulnerability detection. Chapter 4 describes the implementation of the work as well the data structures generated by the PHP-Parser library. Chapter 5 discusses results, limitations and desired future work. Chapter 6 concludes the paper.

Related Work

A number of authors have utilized static taint analysis to identify vulnerabilities and shown that it can be an effective tool for source code vulnerability detection. Huang et al.[12] created the Web application Security by Static Analysis and Runtime Inspection (WebSSARI) tool which utilized static analysis to identify application vulnerabilities. Rather than focusing on control flow as many previous authors had, the authors instead utilized information flow as a mechanism for finding vulnerable code paths. An AST was used as the intermediate representation, with support for included files during the AST generation. A control-flow graph and symbol table were utilized to allow the analysis engine to determine types for variable and function pre- and post-conditions. Following this, the AST was evaluated for insecure data flow paths. WebSSARI was designed to be modular such that different parsers and lexers could be written and utilized permitting different languages to be analyzed, though the authors focused their efforts on PHP.

Jovanovic et al.[13] used static analysis in their Pixy PHP analysis tool, which was capable of automatically detecting XSS and SQLI vulnerabilities using taint flow analysis. The program utilized an AST as the intermediate representation which was generated by the open-source PHPParser library. The authors used transfer functions to track the propagation of taint through data structures, and handled some major problems inherent with static analysis of PHP code, including variable references, literal value analysis, and included files as well as emulation of built-in PHP functions. The authors did not add support for object-oriented PHP code, and use of member variables and object functions were always

assumed to be untainted.

Gauthier and Merlo[14] developed the Access Control Models Analyzer (ACMA), a tool which tacked a different problem from the OWASP Top 10[2]: Access Control Vulnerabilities. While the vulnerability is different from other works reviewed in this paper, some of the approaches and techniques are similar. The authors utilized both a CFG generated by a parser, which was itself generated by JavaCC, as well as an AST. They compared taint analysis to access control model checking in situations where one boolean privilege exists and drew from this methodology when developing their control flow taint analysis and suggested that sanitization functions are analagous to access control routines in their function (preventing unauthorized actions). The program successfully identified a number of access control vulnerabilities, performing almost 900 times faster than existing solutions.

Zhao and Gong[1] developed a system which utilizes the AST intermediate representation (IR) generated by the HipHop Virtual Machine (HHVM)[1]. The authors converted the code into the AST IR using the HHVM, created a control flow graph (CFG) and performed static analysis on the AST. The data structure used in tracking was detailed and provided an interesting way of storing the relation between variables in a tree-like data structure consisting of storage and index nodes. This static analysis was followed up with dynamic analysis which utilized a URL crawler to identify code paths. These URLs were then deconstructed to create lists of values for known parameters which allowed the code analysis to target known valid code paths. Fuzz testing was then performed by reconstructing various parameter lists, fixing some parameters with known values, and fuzzing the non-fixed parameters. The program was tested against 55 custom vulnerable code samples (each with a corresponding ”fixed” sample) which were designed to reflect specific constructs in the PHP language. The results showed a high (80%) rate of detection for specific language elements but only a moderate (44%) rate of detection for specific injection attacks.

Dahse and Holz[15] developed a precise static analysis tool named RIPS which can

[1] http://hhvm.com/

detect new vulnerabilities in PHP code. RIPS is an open-source PHP static analysis project which was designed to automatically detect 15 types of common vulnerabilities. Unlike many other tools it was developed to model PHP built-in functions and simulates execution of these functions for a more complete analysis. It provides a GUI in the form of a web interface as well as graphical representations of control flow similar to those found in commercial programs such as IDA Pro[2]. Along with these novel features, the authors defined over 230 sensitive sinks in the application and added support for file inclusions. Additionally, RIPS provides an exploit creator which can assist researchers in designing proof-of-concept (PoC) exploits. However, RIPS only supports PHP 4 and lacks support for OOP. As these limitations were apparently caused by fundamental limitations in the design, the program was abandoned and subsequently completely rewritten and is now a commercial offering[16].

Nunes et al.[17] designed the phpSAFE system to detect XSS and SQLI vulnerabilities in PHP-based plugins for web frameworks, specifically targeting the content management system (CMS) WordPress. The authors highlight the lack of OOP support in various other implementations of PHP static analysis tools like Pixy and RIPS, as this prevents these programs from analyzing code from the most-used PHP web frameworks as these frameworks are written with objects. With this in mind, Nunes et al. designed phpSAFE specifically targeting support for OOP and handling of CMS plugins. The authors used data flow analysis to track the propagation of tainted data through the program. phpSAFE differs from previous implementation of PHP static analysis tools as it handles specific cases that occur with plugin code, such as files with no main function.

Medeiros et al.[18] evaluated PHP code for input validation vulnerabilities using static analysis, also using taint flow analysis to track the spread of untrusted input through an application. Their approach used analysis of an AST in conjunction with data mining, code correction and user feedback to the developers to show vulnerable code along with the sug-

[2]https://www.hex-rays.com/products/ida/index.shtml

gested correction. As the authors pointed to several circumstances in which it is difficult to determine whether or not a vulnerability exists; for example, in a substring operation on a tainted string, it uncertain whether the new string should be tainted. Machine learning was incorporated in conjunction with data mining to reduce the rate of false positives by providing a machine learning algorithm with samples of code which humans classified as vulnerable or not vulnerable, allowing the system to reduce the number of false positives based on certain ”symptoms” in the code. A wide range of vulnerabilities were supported for detection, from XSS and SQLi vulnerabilities to remote file inclusion, local file inclusion, directory traversal, source code disclosure, PHP code injection and operating system command injection.

Bandhakavi et al.[19] targeted the JavaScript language with their VEX framework, which was designed to analyze Mozilla Firefox extensions for patterns that could result in information flow vulnerabilities. The authors identified five specific types of information flow from sources to sinks that constitute patterns which may result in a vulnerability. In reviewing JavaScript extensions, it was noted that they tend to be very complex. The extensions generally have a significant number of functions and rely heavily on OOP as well as utilize dynamic programming extensively. The authors proposed an abstract heap design for their program, which permits accurate tracking of the relationships between objects and functions by utilizing a graph. An AST was used as the code intermediate representation, and the VEX program walked the AST to create the abstract heap. The authors note that VEX found three previously unknown vulnerabilities in JavaScript extensions, though posit that adding pointer analysis in future revisions could enhance the precision of the program.

Design

A number of challenges presented in the design of this program. Below we detail these challenges and the solutions which were developed for solving them.

3.1 Problem Formulation

In this work, we propose a system that can identify XSS vulnerabilities in PHP code. The basis for finding these issues lies in the analysis of data flow through code. There exist a number of "sources" and "sinks" through which tainted data can flow in a PHP application. Our method of static analysis involves converting the source code to an intermediate representation which may then be more readily interpreted.

3.2 Intermediate Representation

It was determined that program analysis would be simplified through the use of an intermediate representation. We elected to utilize an abstract syntax tree for this purpose due to the ease with which it can be converted to and from source code. Additionally, the tree structure lends itself well to a recursive analysis algorithm, which simplifies some of the challenges with handling scopes for multiple branches. The AST is constructed of nodes which may be either statements or expressions. Nodes may or may not have children; if any children exist, those must be recursively analyzed in a depth-first manner before the result of the parent node can be determined.

3.3 Syntax Rules

Interpretation of the program requires a set of rules which describe the language. The behavior of each operation is modeled based on a set of rules which specifies the results from and impacts of supported operations. The list of these operations in the PHP language is given in Table 3.1 below. This list was selected so as to cover the most commonly-used elements of the PHP programming language.

Table 3.1: PHP Syntax

Expressions ::=	
$\mid x$	variable
$\mid x[e]$	array index access
$\mid e\ op\ e$	binary operation
$\mid function(p_1, ..., p_n)$	function definition
$\mid f(a_1, ..., a_n)$	function call
Statements ::=	
$\mid break$	break
$\mid return\ e$	return
$\mid x := e$	assignment
$\mid if\ e\ then\ S_1\ else\ S_2$	conditional
$\mid while\ e\ then\ S$	while
$\mid do\ S\ while\ e$	do-while
$\mid for(S_1;\ e;\ S_2)\ do\ S_3$	for
$\mid foreach\ e\ then\ S$	foreach
$\mid switch\ e_0\ case\ e_1\ then\ S_1$	switch

3.4 Taint Flow Analysis

Tracking the flow of tainted data through the program is paramount to the task of identifying XSS vulnerabilities. In this work, we use taint flow analysis to record where and how tainted data flows in a program so that we can reliably detect when sensitive operations are performed on it. This process begins with identifying ways in which taint is generated, propagated and removed. There are three major classes of operations that can result in the input, output or removal of taint from the system: sources, sinks and sanitizers (respectively).

3.4.1 Sources

Sources are code elements that are assumed to generate untrusted input to a program. Untrusted inputs could result from reading data from a file (which may be changed outside of the program), accepting user input (which may have malicious content) or referencing data that was passed into the program via certain superglobal[1] variables (some of which can be set by clients). In the case of this program, we consider only the last case, as those receive input directly from the user's browser. Of the superglobal variables, several are implicitly considered tainted due to their being set based on values returned from the client browser: $_GET, $_POST, $_FILES, $_COOKIE, $_REQUEST and $_SERVER.

The $_GET array receives its values from URL parameters passed in with a query, which could be malicious if a user clicks an untrusted link with malicious parameters attached. $_POST contains data passed in through POST requests (typically form data), which could potentially include script data. $_FILES is an associate array of items uploaded with a POST request which contains string fields that may be manipulated. $_COOKIE can be set based on prior user input and may also be tainted. $_REQUEST by default contains the contents of $_GET, $_POST and $_COOKIE, all of which are considered unsafe.

[1] http://php.net/manual/en/language.variables.superglobals.php

$_SERVER contains information such as the request URL, which may be script and could pose a threat if displayed on the page.

For our purposes, we consider any reference to a key within one of these arrays to be tainted. When an expression is evaluated for taint and one of these sources is present, it results in the entire expression being considered tainted and the taint is propagated via the rules listed in the previous section.

3.4.2 Sinks

There are sensitive operations in PHP code where the use of untrusted data could result in a XSS attack. In this particular case we are specifically concerned with operations that can display information on a web page, as this could result in malicious script being executed in a client browser. There are a number of functions that output to the page, however for the purpose of simplicity we only cover the non-formatted string output functions, **echo()** and **print()**. Both functions take expressions as arguments, however the print function only allows one argument while the echo function allows any number of arguments.

Upon encountering either of these sinks being utilized in a program being analyzed, their expression arguments are evaluated recursively to determine if any taint sources are being used to create the output. If any are found, the taint flow history from that source is retrieved and displayed to the user of this program, presenting them with a textual representation of how the tainted information flowed from a source to this sink.

3.4.3 Sanitizers

There are a number of ways in which taint can be removed from a variable or expression. Regular expressions can be used to filter out special characters, custom functions can be written that encode script tags and PHP built-in functions for escaping special characters and encoding strings can all be used to sanitize output to prevent XSS attacks.

Unfortunately, it would be an entire work in itself to support every possible mechanism

of taint sanitization as programmers might write their own custom sanitizer functions or regex matching strings and handling these could require writing a regex interpreter, which would be outside the scope of this project. A reasonable compromise was chosen in which a number of built-in functions that are capable of preventing script from being output to the page are recognized by default, with the option to manually add additional functions by name to this list.

The two PHP built-in functions recognized are **htmlspecialchars** and **htmlentities**, which HTML encode special characters. When a sanitizer function is invoked in code being analyzed, a function is called which performs actions based on how that function operates. In the case of the two functions just listed, the function immediately returns False to indicate that the return from these functions is considered untainted. Other, more specialized functions could be added to this list and handled appropriately allowing for a high level of flexibility in handling their emulation and impact on data flow.

3.4.4 Approximation

Due to the nature of static analysis reducing the flow of a program to a single execution path, any results generated from statements that result in a multiple code branches must be condensed into a single result at the end of that basic block. In this work, we chose to analyze each branch individually then merge the results at the end of the split. As each branch is analyzed, the statements are evaluated within the context of a current *environment*.

Each branch begins with an environment that contains all variable and taint history from the actions leading up to the conditional, however changes that occur do not effect any other branch. At the end of the branching, a logical OR operation is performed on the taint value of a given variable across all possible branches; if any branch contains that variable in a tainted state, then that variable is tainted as a result. This ensures that any taint of variables is preserved at the end of the branching, regardless of from which branch it originated. Listing 3.1 below shows an example of this approach.

Listing 3.1: Taint Overapproximation

```
1 $a = '';
2 $var = '';
3 if(condition1){
4     $var = $_POST['data'];
5 } elseif(condition2) {
6     $var = 'safe';
7 }
8 echo $var;
```

In this simple example, at the time of execution the current environment is δ, which contains simply the variable $var. The first branch of the conditional works on a copy of the environment, δ', and then stores into the variable $var tainted data from PHP's $_POST superglobal array, which contains parameters passed along with the POST request. This environment is retained until the end of the conditional so that it may be compared with the others. In the elseif branch, a separate copy of the environment is made, δ'', and $var is assigned a string literal which is considered untainted. However, at the end of the conditional, there will only be one execution path.

To accommodate this scenario, the environment from each conditional is compared with the others. Any variables declared in the branches would go out of scope at the end of execution of the branch, and are subsequently discarded. For each variable from the parent environment δ, the tainted status of that variable from all environments δ, δ' and δ'' are used in a logical OR operation such that if any environment contains a tainted version of the variable, then the result at the end of the conditional indicates that the variable is now tainted going forward. If more than one branch tainted the variable, the line numbers of all taint sources are recorded so that the user knows that multiple locations generated the tainted data. This is implemented via the $merge$ function, which accepts as arguments three parameters: the parent environment δ_p, the branch environment δ_b, and a boolean representing whether at least one branch of the conditional must execute.

The boolean parameter in the $merge$ function handles a specific case that can arise with multiple branches:

- A variable is tainted before entering a conditional construct
- The conditional has a default condition
- Every branch results in the variable being in an untainted state at the end of execution

In this case, due to the presence of a default statement which guarantees at least one branch will be executed, we can safely say that the variable will be untainted at the end of the branching. An example of this is given in Listing 3.2 below. The variable $var is tainted on the first line, however since all branches in the conditional result in $var being untainted and the presence of a default condition (an else statement) ensures execution of at least one branch, it is guaranteed that the variable will no longer be tainted at the end of the conditional block.

Listing 3.2: Branch Execution with Default

```
1 $var = $_GET['data'];
2 if(condition1){
3     $var = 0;
4 } elseif(condition2) {
5     $var = 1;
6 } else {
7     $var = 2;
8 }
9 echo $var;
```

The analysis of PHP code in this project was not performed on the source code directly, but rather on an AST generated from the source code. This format succinctly represents the structure and logic of the code while abstracting away the specific syntax of the language, making the program analysis simpler as the resulting tree is significantly easier to work with than source code. The open source library PHP-Parser[2] was used in this project to provide this functionality. PHP-Parser generates an AST of nodes that can be evaluated more easily, in this case using a recursive approach.

[2]https://github.com/nikic/PHP-Parser

The PHP-Parser library defines three types of nodes; statements, expressions, and scalars. Statement nodes are those which do not return a value. Function definitions, loops, and conditionals are all considered to be statements. Statements can occur as a result of other statements; for example, an *if* statement can contain a series of statements that execute as a result of the condition evaluating to true.

Expressions are nodes that return values, such as variables, boolean operators, concatenation operations, array indices, and assignment operations. Expression nodes can occur inside other expressions; for example, if we let $a = b||(c||d)$, the resultant node is an expression node which contains an inner expression node which must first be evaluated before the result of the outer expression can be evaluated.

Scalars are a special case of an expression, representing things like hard-coded integers, strings and predefined constants called ”magic constants”. These are generally assumed to be safe from tampering, as they should not have an unsafe value by default. The value of these could be manipulated after the initial assignment to a variable, however such manipulation would be assumed to be safe barring the use of any tainted data in that process.

As an example, in Figure 3.3 below a small section of PHP code is shown that retrieves a value from the superglobal $_POST array and stores it to a variable $var before finally echoing it to the page as part of a concatenated string. Figure 3.4 below shows the AST that is generated from this code. In this example, the assignment expression Expr_Assign stores the result of the expression Expr_ArrayDimFetch to the variable given by the expression Expr_Variable. The result from the Expr_ArrayDimFetch is dependent on the result of two more nodes, an Expr_Variable and an index into that variable given by the Scalar_String (which is also an expression).

The second operation is somewhat more complex. The Stmt_Echo accepts an array of expressions, in this case an array containing one item. The only entry is a concatenation operation, the Expr_BinaryOp_Concat. This expression concatenates the result of another

concatenation operation with the Scalar_String whose value is simply an exclamation point.

Listing 3.3: Example Code

```
1 $var = $_POST['input'];
2 echo "You entered " . $var . "!\n";
```

Listing 3.4: Example AST

```
1  array(
2      0: Expr_Assign(
3          var: Expr_Variable(
4              name: var
5          )
6          expr: Expr_ArrayDimFetch(
7              var: Expr_Variable(
8                  name: _POST
9              )
10             dim: Scalar_String(
11                 value: input
12             )
13         )
14     )
15     1: Stmt_Echo(
16         exprs: array(
17             0: Expr_BinaryOp_Concat(
18                 left: Expr_BinaryOp_Concat(
19                     left: Scalar_String(
20                         value: You entered
21                     )
22                     right: Expr_Variable(
23                         name: var
24                     )
25                 )
26                 right: Scalar_String(
27                     value: !
28 
29                 )
30             )
31         )
32     )
33 )
```

This is a simple example that demonstrates the structure of the AST and how it relates to the original PHP source code. More complex samples have been utilized to test this program, as vulnerable PHP code in the wild is much more elaborate.

3.5 Evaluating Nodes

Based off the syntax rules established previously, we have defined semantic-based inferences that describe how nodes are evaluated. Tables 3.2 and 3.3 below detail these inferences.

Table 3.2: Expression Syntax Rules

Syntax Rule	Semantic-Based Inference
x	When a variable is used in an expression, the taint status of the variable is returned so that the expression as a whole may be evaluated as a combination of each of the parts
$x[e]$	Array index accesses are checked against the variable declaration data structure; the key can be either a scalar value or a variable
$e_1 = e_2 \, binaryop \, e_3$	Binary operations are performed by evaluating each of the operands and performing a logical OR operation of their results; if either of the operands is tainted, then the result is tainted
$f[name] = function(p_1, ..., p_n)S$	Function definitions store the entire contents of the function definition in an array with an index of the function name; this permits easy access to evaluate the code later when that function is called
$function(p_1, ..., p_n)$	Function calls receive the taint status of each argument and analyze the statements of the function in an environment that reflects the taint status of those arguments

Table 3.3: Statement Syntax Rules

Syntax Rule	Semantic-Based Inference
$break$	Break statements prevent control fallthrough in switch statements, resulting in analysis of the next case statement in line (break statements in loops are ignored to ensure all statements inside the loop are processed)
$return\ e$	Return statements evaluate the expression provided after the statement for taint; the result of this evaluation is set as the returned taint status from the function call
$x = e$	The expression being assigned to the variable is evaluated; if e is untainted then x is marked untainted and the taint history of x is cleared, otherwise x will be marked tainted and a unique join of the taint history of each component in e is stored as the taint history of x
$if\ e_1\ then\ S_1\ elseif\ e_2\ then\ S_2\ else\ S_3$	The statements S in each conditional block are evaluated in their own scope, and modify only their own scope; the results are merged at the end and can taint or untaint variables in the parent environment
$while\ e\ then\ S$	The statements S are evaluated in a new scope; at the end of the loop, any local variables are discarded and any newly-tainted variables are marked as such in the parent scope
$do\ S\ while\ e$	The statements in a do-while loop are evaluated similarly to the while loop above
$for(S_1; e; S_2)\ then S_3$	The initialization (S_1) and afterthought (S_2) statements of a for loop are evaluated in a new scope, after which the body of the loop is evaluated; at the end of the loop, local variables are discarded and newly-tainted variables are marked tainted in the parent scope
$foreach(e_1\ as\ e_2)\ do\ S$	If any items in e_1 are tainted, then e_2 is marked tainted in the local scope; the statements in the loop are then evaluated, local variables are discarded and newly-tainted variables are marked in the parent scope
$switch(e)\ case\ e_n :\ S_n$	Each case is evaluated in turn in its own scope, continuing until a break is found or the switch ends. Locals are then discarded and any new taint is merged back into the parent scope.

The $eval$ function contains the logic for handling each type of node. It accepts as an

argument a node from the AST which may be either a statement or an expression. When evaluating expressions the function will return a boolean which indicates the taint status of the expression, whereas evaluation of statements will not return a value but can modify the various data structures.

Implementation

The goal of this project was to implement support for a core set of functionality in the PHP language (Figure 3.1 in Chapter 2 shows the supported code elements). In this work, we use static analysis of PHP code to find vulnerabilities which could allow an XSS attack against a web site. Additionally, we present any identified issues to the user so that the vulnerabilities may be remedied.

4.1 Variable Taint

The taint status of variables is tracked in a map data structure where the key is the name of the variable, and the value is an array of attributes of that variable, specifically the taint status (recorded as a boolean) and the taint flow history of that variable (represented as an array of line numbers through which the taint was propagated). This structure is illustrated in Listing 4.1 below.

Listing 4.1: Variable Taint Data Structure

```
vars = array(
    'key_1' => array(
        'tainted' = <bool>,
        'taint_flow' = array(
            <integer_1>,
            <integer_2>,
            ...
            <integer_N>
        )
    ),
    'key_2' => array(
        'tainted' = <bool>,
        'taint_flow' = array(
            <integer_1>,
            <integer_2>,
            ...
            <integer_N>
        )
    )
)
```

This structure allows us to not only track whether or not a variable has been contaminated with untrusted input, but also to store a history of lines through which the tainted data has flowed to reach a given variable. Storing this data allows us to later show the user visually where the vulnerability occurred to give them a better idea of the conditions that led to the unsafe usage of tainted data.

A simple example of this data structure follows, in order to demonstrate how the data is structured. In the case of the code shown in Listing 4.2 below, the following Table 4.1 results:

Listing 4.2: Sample Taint Generation

```
$my_arr['val'] = 0;
$b = $_GET['val'];
$var = $b;
```

Table 4.1: Variable Taint Data Structure

Variable Name	Tainted	Taint Flow
my_arr['val']	False	NULL
b	True	2
var	True	2,3

In this example, the array index $my_arr['val'] is assigned a scalar value, which is assumed to be untainted. The variable $b then receives data from a taint source, and both the new taint status and the line on which the taint originated are stored with the variable. Then $var receives the content of $b, which results in taint propagating to this new variable. In addition, when considering the new taint history of $var, an array is built containing the taint history of all expressions used in this new assigned value (in this case just line 2). We then add the current line 3 to this array and trim the array to ensure only unique values, in case two expressions have taint that flows through the same line.

This approach allows us to record not only the immediate source of variable taint but also the history of any operations through which this taint has been transmitted until it arrived at the new variable assignment. Once the tainted variable $var is used at a sensitive sink, the program can then display to the user exactly how the tainted data flowed through the program to get to $var and provides a clearer picture of what steps need to be taken to remedy this situation.

As can be seen in Table 4.1, standard scalar variables are represented simply by their name, whereas an index into an array is stored as the array name along with the key. An interesting issue arises due to the nature of array index assignments in PHP, as PHP arrays are actually ordered maps. Note that this poses issues with array access by integer indices, something which will be discussed further later in the paper along with potential solutions.

4.2 Function Definitions

In order to facilitate the handling of function calls in the code being analyzed, a map was utilized which stored the code defining the function. The name of the function was used as the key which maps to the function code, allowing easy access to the function code whenever a call to that function was made for the purpose of evaluating the function code with the given parameters. An initial pass through the file fills this array as PHP does not require functions to be defined before use; this ensures that functions defined at the end of a file are available to be analyzed from calls made prior to the function definition.

4.3 Taint Propagation

After the definitions of taint flow were defined and the data structures created, we needed to translate the syntax rules for each code construct into the analysis code in the program. Identification of different node types was performed using a switch statement on the class of a node (retrieved from the PHP built-in function get_class), and the fully-qualified class names were used for the case statements; for example, the Assignment operation has a node class of $PhpParser\backslash Node\backslash Expr\backslash Assign$.

At the beginning of execution, two global variables are initialized to empty arrays: $vars and $functions. The former is the data structure where variables will be marked as tainted and the history of their taint flow recorded, while the latter will contain the nodes representing function definitions for easy access later. Additionally, a number of flag variables were set that allow tracking of specific conditions, such as $has_return which is True if a function returns data, $returns_taint which is True if a function's return statement evaluates to True (or rather is tainted), and $break_hit, which identifies when a switch statement pass has hit a break statement.

4.3.1 Taint Propagation Functions

There are a number of important functions that assist in managing the propagation of taint from nodes. The merge_vars_arrays function takes two copies of the $vars array; the first copy is an updated copy (or a new scope) we'll call a child array, the second is the parent from which the child was derived. Remember that the keys in these arrays are a variable name, and the values are an array which contains the tainted status of the variable as well as line numbers through which the taint flowed. The function then iterates through each key (a variable name) in the parent array and checks if the key exists in the child. If it does not, it adds the key and value from the parent array to a new result array. If the value does exist in both, the key is stored to the result array and the value is calculated by calling merge_vars_props on the values from both the parent and child array. This continues until all keys have been processed.

The merge_vars_props function performs the merge on the properties of these tainted variables. The function determines the new ”tainted” status of the variable as the union of the taint status from the parent and child array values. The taint flow history from both the parent and child $vars arrays are merged, keeping only unique values; this lets us track any new propagation in this child array.

Additionally, both the merge_vars_arrays and merge_vars_props functions accept an optional third argument, $mandatory, which states whether or not the child array is from a code segment which is guaranteed to be executed. For example, in a do-while loop, it is guaranteed that the code will be executed at least once, so when calling merge_vars_arrays the flag is set to True (which will be passed subsequently to merge_vars_props). In this way, if a variable is tainted before a code section which is guaranteed to execute at least once and then untainted at the end of the block, we can safely assume that the variable will be untainted after the block has completed.

4.3.2 Hard-coded Expressions

The simplest nodes to evaluate are the String scalar, the ConstFetch expression (for using a defined constant), and the InlineHTML nodes. These are all defined by the programmer and hard-coded in the application, therefore they are presumed to be untainted as we assume the developer does not have malicious intent. As a result, when evaluated these nodes will always return a value of False.

4.3.3 Variable Assignments

Evaluating the assignment of values to variables is critical in tracking taint propagation. The assignment operation in PHP is unique in that it is a statement but also behaves like an expression as it returns a value. For example, $\$a = (\$b = 4) + 5$ is a valid statement in PHP and results in $b being set to 4 and $a being set to 9. We support this usage of the assignment operator for completeness.

In the example Listing 4.3 below, we illustrate the flow of taint from the $_POST superglobal variable to the variable $a. From there the taint transfers to $b before being used in an echo statement. The implementation for supporting this operation is shown in Listing 4.4 below. First the name of the variable is retrieved; if the assignment is not directly to a variable then the $var_name function composes the variable name (an example being an index into an array, such as $arr[1]), otherwise the name is set to the variable name. If the target variable is not yet in the $vars array, the data structure for the variable is initialized. The expression being assigned is then evaluated, and the taint status and taint flow history are transferred to the target variable name.

Listing 4.3: Assignment Operation Example

```
1 $a = $_POST['name'];
2 $b = $a;
3 echo "Hello, " . $b;
```

Listing 4.4: Assignment Operation Evaluation

```
case "PhpParser\Node\Expr\Assign":
    // Get the name of the variable to which we're assigning
    $var_name = '';
    if(!($node->var instanceof PhpParser\Node\Expr\Variable)){
        $var_name = get_var_name($node->var);
    } else {
        $var_name = $node->var->name;
    }

    if(!array_key_exists($var_name, $vars)){
        init_var($var_name);
    }

    if(eval_node($node->expr)){
        taint($var_name);
        if(is_a($node->expr, "PhpParser\Node\Expr\FuncCall")){
            $history = $returns_taint_history;
        } else {
            $history = get_taint_flow($node->expr);
        }
        $flow_to_add = array_merge($history, array($currentLine));
        add_taint_flow($var_name, $flow_to_add);
    } else {
        // Remove taint and clear taint_flow array
        untaint($var_name);
        clear_taint_flow($var_name);
    }

    return;
```

4.3.4 Binary Operations

Binary operations are also trivial to evaluate. Listing 4.5 shows an example of a binary operation resulting in tainted output; the concatenation of a hard-coded string along with a value received from a tainted source results in a new string, which is now tainted. This string is subsequently used in a sensitive sink, resulting in an XSS vulnerability. If either operand in a binary operation is tainted, then the result is considered tainted. As all binary operations follow this pattern, a regular expression was used to trigger evaluation of this case. Note that the body of the switch statement has been included in Listing 4.6 below to show how the ternary operator in the case statement triggers execution.

Listing 4.5: Binary Operation Example

```
1 $name = $_POST['username'];
2 echo "Hello, " . $name . "\n";
```

Listing 4.6: Binary Operation Evaluation

```
1 switch($node_class){
2     ...
3     case (preg_match("/BinaryOp/", $node_class) ? $node_class : ...
        !$node_class):
4         if(eval_node($node->left) or eval_node($node->right)){
5             return True;
6         }
7         return False;
8     ...
9 }
```

4.3.5 Loops

Loops are also relatively easy to evaluate. An example of a loop resulting in an XSS vulnerability is shown in Listing 4.7 below, which repeats a user-entered string five times. Processing a while loop (show in Listing 4.8) for taint flow begins by saving a copy of the current global $vars array, so that we are able to identify later which variables were defined in the scope of this loop (or local variables). The condition is then evaluated, as assignment operations can occur as part of the conditional expression. Then each statement in the body of the while loop is recursively evaluated to determine taint propagation. At the end of the loop, the merge_vars_arrays function (explained above) is called to propagate taint from the loop scope back to the parent scope and to delete local variables.

Listing 4.7: While Loop Example

```
1 <?php
2 // Repeats a string five times
3 $output = '';
4 $ctr = 0
5 while($ctr < 5]){
6     $output = $output . $_POST['str'];
7     $ctr++;
```

```
8 }
9 echo $output;
10 ?>
```

Listing 4.8: While Loop Evaluation

```
1  case "PhpParser\Node\Stmt\While_":
2      $saved_vars = $vars;
3
4      eval_node($node->cond);
5
6      foreach($node->stmts as $stmt){
7          eval_node($stmt);
8      }
9
10     // Merge any updates back in to the main array, remove any locals
11     $vars = merge_vars_arrays($vars, $saved_vars);
12
13     return;
```

The do-while loop is processed in much the same way as the while loop, however the merge_vars_arrays function is called with the $mandatory flag set to True, since the body of the loop will be executed at least once.

For loops consist of three main components with which we are concerned: the initialization, the loop action(s), and the loop statements. An example for loop is given below in Listing 4.9, which shows a simple for loop that initializes the variables $i and $j (the latter being tainted), prints them out on each iteration, then increments $i. The procedure for evaluating this loop is given in Listing 4.10. After saving a copy of the parent $vars array, the initial conditions are evaluated (in our example, this results in $j becoming tainted). The loop action ($i++) is then processed, followed by the statements in the body of the loop. Once all statements have been processed, the results are merged back into the parent array.

Listing 4.9: For Loop Example

```
1  <?php
2  for($i = 0, $j = $_POST['j']; $i < 10; $i++){
3      echo $i;
```

```
4      echo $j;
5  }
6  ?>
```

Listing 4.10: For Loop Evaluation

```
1  case "PhpParser\Node\Stmt\For_":
2      $saved_vars = $vars;
3
4      // Evaluate the initial condition(s)
5      foreach($node->init as $stmt){
6          eval_node($stmt);
7      }
8
9      // Check the loop action(s)
10     foreach($node->loop as $stmt){
11         eval_node($stmt);
12     }
13
14     // Evaluate the statements inside the loop
15     foreach($node->stmts as $stmt){
16         eval_node($stmt);
17     }
18
19     $vars = merge_vars_arrays($vars, $saved_vars);
20
21     return;
```

The foreach function operates on a somewhat different premise than the other loops. The foreach loop iterates over each entry in an array and uses each value as the loop variable. Listing 4.11 shows an example of a foreach loop where one of the array values is tainted, while the code for handling a foreach loop is given in Listing 4.12. The $vars array is backed up, after which the array to be iterated over is searched to find any tainted values. Since the array indices are stored as array_name[key_name] in the $vars array, we use a regular expression to identify values stored in the array. If any tainted values are found, the whole array is considered to be tainted for the purposes of this analysis (as each value will be used). We then create a variable named as the $value and set it to the taint status of the array found previously, adding any taint history from the tainted variable to this new loop variable. We are then able to evaluate the statements in the foreach loop, merging the variable taint status at the end and removing local variables.

Listing 4.11: Foreach Loop Example

```php
<?php
$arr[0] = $_POST['test'];
$arr[1] = 1;
foreach($arr as $value){
    echo $value;
}
?>
```

Listing 4.12: Foreach Loop Evaluation

```php
case "PhpParser\Node\Stmt\Foreach_":
    $saved_vars = $vars;

    // Find if any items in the target array are tainted
    $tainted = False;
    $flow = array();
    foreach($vars as $var => $values){
        if(preg_match('/^' . get_var_name($node->expr) . '/', ...
            $var) === 1){
            if(is_tainted($var)){
                $tainted = True;
                // Now get the taint flow so we can see where the ...
                    source was
                $flow = array_unique(_get_taint_flow($var) + $flow);
    }    }    }

    // Add our loop variable and set the taint status if needed
    init_var(get_var_name($node->valueVar));
    if($tainted){
        taint(get_var_name($node->valueVar));
    }
    add_taint_flow(get_var_name($node->valueVar), $flow);
    foreach($node->stmts as $stmt){
        eval_node($stmt);
    }
    // Update any changed vars, remove locally scoped vars
    $vars = merge_vars_arrays($vars, $saved_vars);
    return;
```

4.3.6 Conditionals

Both the if-else and switch conditional statements are supported in this project. As with the do-while loop, a scenario occurs with these conditionals that can guarantee at least one branch will be executed. An $else$ statement at the end of an if conditional as well as a

default case in a switch statement both ensure that a minimum of one branch will be taken. Below we detail the use and implementation of these conditionals in our project.

Listings 4.13 and 4.14 below show a sample if-else construct and the code which analyzes these statements. The example below shows how the variable $var is tainted in only the first branch of the if-else conditional but not the others. The evaluation for the if conditional saves the $vars array as standard, but also creates a new $merged_vars array which will save the results of merging each branch with the previous results. We then process each statement inside the if block, merging the results of the temporary $vars array into $merged_var. For each $elseif$ statement, a new $vars array is created from the original parent $vars array and the statements in the $elseif$ block are evaluated in that environment before merging the results into the $merged_vars array. Finally, the $else$ block (if one exists) is processed and the results merged. Once all blocks are processed, the merged variables array is itself merged back with the parent array. As was noted earlier in this chapter, variables can be untainted if the following two conditions are met:

- An $else$ branch exists
- The variable is untainted in each branch of the conditional

If both conditions are met, then it can be safely assumed that every possible outcome results in the variable having taint removed and that there is no way for the conditional block to execute without untainting the variable.

Listing 4.13: If-Else Conditional Example

```
1  $var = '';
2  $cond = 1;
3  if($cond == 0{
4      $var = $_GET['data'];
5  } elseif($cond == 1){
6      $var = 'stuff';
7  } else {
8      $var = 'things';
9  }
10 echo $var;
```

Listing 4.14: If-Else Conditional Evaluation

```
case "PhpParser\Node\Stmt\If_":
    $saved_vars = $vars;
    $merged_vars = array();

    // Evaluate the initial if condition statements
    foreach($node->stmts as $stmt){
        eval_node($stmt);
    }

    $merged_vars = merge_vars_arrays($merged_vars, $vars);

    // Evaluate each elseif and merge in the taint status and history
    if(!is_null($node->elseifs)){
        foreach($node->elseifs as $elseif){
            $vars = $saved_vars;
            eval_node($elseif);

            $merged_vars = merge_vars_arrays($merged_vars, $vars);
        }
    }

    // Evaluate the else and merge in the taint status and history
    if(!is_null($node->else)){
        $vars = $saved_vars;
        eval_node($node->else);

        $merged_vars = merge_vars_arrays($merged_vars, $vars);
    }

    if(!is_null($node->else)){
        $vars = merge_vars_arrays($saved_vars, $merged_vars, True);
    } else {
        $vars = merge_vars_arrays($saved_vars, $merged_vars);
    }

return;
```

The other conditional construct we handle is the switch statement. Listings 4.15 and 4.16 below show an example of a switch statement in PHP and our implementation of taint support. In the example we assign a value to $result based on which case is executed, though the absence of a *break* statement in the ”first” case causes control fallthrough resulting in the ”second” case being executed as well. If no cases match, the ”default” case is executed.

Support for this construct requires a slightly different approach, as the possibility of fallthrough when *break* statements are not present means that the statements from multi-

ple cases could be executed. In addition to the standard saving of $vars and creating the $merged_vars array, we also save the portion of the AST containing the cases, permitting us to jump back to evaluate previous cases if fallthrough occurs. From this point, we evaluate each case individually, continuing as long as a $break$ statement is not found. Once a $break$ is encountered, the $\$break_hit$ flag is set and execution will not continue to the next case. As with the $if - else$ example, the results of each branch are merged at the end, and the presence of a default case permits variables to be marked untainted so long as all branches result in the variable being untainted.

Listing 4.15: Switch Example

```
1  $var = "second";
2  $result = '';
3  switch($var){
4      case "first":
5          $result = "First case executed";
6      case "second":
7          $result = "Second case executed";
8          break;
9      default:
10         $result = $_REQUEST['data'];
11 }
12 echo $result;
```

Listing 4.16: Switch Evaluation

```
1  case "PhpParser\Node\Stmt\Switch_":
2      $saved_vars = $vars;
3      $merged_vars = array();
4      $cases = $node->cases;
5      $is_default = False;
6
7      // Evaluate each case statement sequentially.
8
9      for($i = 0; $i < count($cases); $i++){
10         // A null condition means this is the "default" case
11         if(is_null($cases[$i]->cond)){
12             $is_default = True;
13         }
14
15         for($j = $i; $j < count($cases) && !$break_hit; $j++){
16             // Evaluate the statements for this case statement
17             foreach($cases[$j]->stmts as $stmt){
```

```
18                      eval_node($stmt);
19                  }
20              }
21              $merged_vars = merge_vars_arrays($merged_vars, $vars);
22
23              // Reset the vars array
24              $vars = $saved_vars;
25
26              // Reset the break flag
27              $break_hit = False;
28          }
29
30          if($is_default){
31              $vars = merge_vars_arrays($saved_vars, $merged_vars, True);
32          } else {
33              $vars = merge_vars_arrays($saved_vars, $merged_vars);
34          }
35
36  return;
```

4.3.7 Functions

Our program also supports taint analysis of functions defined in the target code. The initial function definition is handled by the function data structure outlined previously in this document. Prior to starting analysis of the code, the AST is traversed searching for any function definitions (as PHP does not require functions to be defined before use). Any that are found are added to the function datastructure, where the key is the function name and the value is the AST node corresponding to the function code. At the time a function is called, the name of the function is checked to see if it is a built-in sanitization function (other built-in functions are currently ignored); if it is, a False value is returned. If the function is in the list of user-defined functions, the arguments are each analyzed to determine if they are tainted. A new scope is set up in which the function's parameters are tainted accordingly to the argument taint status, all other variables are removed (since this is a local function scope), and the statements in the function are analyzed. If the function has a $return$ statement, a taint status of the expression from the $return$ node is determined and returned as the result of the function call. At this point, local variables from the scope of the function are removed.

Listing 4.17 shows an example of a defined function, its use and return of tainted data while Listing 4.18 shows the code whose operation is outlined above. Comments have been omitted for brevity, however the full text of the code can be found in Appendix A.

Listing 4.17: Function Example

```
1  function gen_taint($val1, $val2){
2      $val3 = $val1 . $_POST['taint'];
3      return $val3;
4  }
5  $var = gen_taint('0', '1');
6  echo $var;
```

Listing 4.18: Function Evaluation

```
1  case "PhpParser\Node\Expr\FuncCall":
2      $has_return = False;
3      $returns_taint = False;
4      $func_name = $node->name->parts[0];
5      $args = $node->args;
6      if(in_array($func_name, $special_functions) || ...
           function_exists($func_name)){
7          $has_return = True;
8          $returns_taint = handle_function($func_name, $args);
9      } else {
10         $saved_vars = $vars;
11         $tmp_vars = array();
12         if(!array_key_exists($func_name, $functions)){
13             return;
14         }
15         $params = $functions[$func_name]->params;
16
17         for($i=0; $i<count($args); $i++){
18             // Get name of param
19             $var_name = $params[$i]->name;
20             $tmp_vars[$var_name] = array(
21                 'tainted' => eval_node($args[$i]->value),
22                 'taint_flow' => get_taint_flow($args[$i]->value),
23             );
24
25             if($tmp_vars[$var_name]['tainted']){
26                 array_push($tmp_vars[$var_name]['taint_flow'], ...
                       $currentLine);
27             }
28         }
29         $vars = $tmp_vars;
30         $stmts = $functions[$func_name]->stmts;
31         foreach($stmts as $stmt){
```

```
32                  eval_node($stmt);
33              }
34              $vars = merge_vars_arrays($vars, $saved_vars);
35          }
36          if($has_return){
37              return $returns_taint;
38          } else {
39              return;
40          }
```

4.3.8 Sinks

There are two sinks we consider for our discussion of reflected XSS attacks; the *echo* and *print* statements. Both of these statements write data out to a client's browser and are locations where malicious code could be injected onto a page. We will discuss only the *echo* statement, as both *print* and *echo* are functionally equivalent with the exception that the *print* statement accepts only one argument while *echo* accepts a list. Listing 4.19 illustrates a simple use of the *echo* statement which retrieves a name from the user and prints it out onto the page. Listing 4.20 shows how *echo* statements are evaluated. The statement node contains a list of expressions, each of which are evaluated to check for taint. If any of the expressions are tainted, the taint history from the tainted node is gathered for display and a message is printed to the user detailing from where the taint originated, where it flowed and the current line number with this sink.

Listing 4.19: Sinks Example

```
1 $name = $_POST['name'];
2 echo "Hello, " . $name;
```

Listing 4.20: Sinks Evaluation

```
1 case "PhpParser\Node\Stmt\Echo_":
2     foreach($node->exprs as $expr){
3         if(eval_node($expr)){
4             echo "\neval_node: Found taint in echo\n";
5
```

```
6               $history = get_taint_flow($expr);
7               $history[] = $currentLine;
8               $history = array_unique($history);
9
10              printf("%5s| %s", "Line", "Content\n");
11
12              foreach($history as $line){
13                  printf("%5d| %s\n", $line, trim($lines[$line-1]));
14              }
15
16              echo "\n\n";
17              return;
18          }
19      }
20      return;
```

Evaluation

This thesis work was evaluated using samples of PHP code that are known to contain XSS vulnerabilities. The output of the program was compared with expected results from these known samples to assess the accuracy of the experimental results.

5.1 Samples

Several real-world code examples with known vulnerabilities were pulled from open source repositories for testing. Finding samples of vulnerable PHP code to use for the analysis portion of this project proved challenging. In deciding to not support OOP in this version, much of the publicly available code became unsuitable for use in testing as most modern PHP programs are written utilizing classes. Therefore, we had to manually craft code samples that would be representative of real-world code resulting in XSS vulnerabilities. Over a dozen samples of varying complexity were created to test the efficacy of this system. To ensure coverage of features, samples were used that included loops (for, while, do-while), conditionals (if, if-else, if-elseif-else), function declarations, function calls (both built-in and declared) and nested conditionals and loops. These examples may be relatively short compared to actual application code that would be found on a web server, however they have been written so as to validate the core functionality of the project.

Listing 5.1 in Appendix B contains vulnerable code pulled from a now-defunct Wordpress plugin. Our program identified eight locations in the code where XSS vulnerabilities could occur and which might require closer inspection. The output of the program is shown

below in Listing 5.1.

Listing 5.1: Example 1 Program Output

```
1
2 eval_node: Taint propagated on line number 9
3
4 eval_node: Found taint in echo
5  Line| Content
6     9| $fasc_plugin_ver = $_GET['ver'];
7    31| <script language="javascript" type="text/javascript"
8         src="../../wp-includes/js/tinymce/tiny_mce_popup.js
9         ?ver=<?php echo $fasc_plugin_ver; ?>"></script>
10
11
12
13 eval_node: Found taint in echo
14  Line| Content
15     9| $fasc_plugin_ver = $_GET['ver'];
16    32| <link rel="stylesheet" href="../../css/button-styles.css
17         ?ver=<?php echo $fasc_plugin_ver; ?>" />
18
19
20
21 eval_node: Found taint in echo
22  Line| Content
23     9| $fasc_plugin_ver = $_GET['ver'];
24    33| <link rel="stylesheet" href="../../css/font-awesome.css
25         ?ver=<?php echo $fasc_plugin_ver; ?>" />
26
27
28
29 eval_node: Found taint in echo
30  Line| Content
31     9| $fasc_plugin_ver = $_GET['ver'];
32    34| <script src="jquery.minicolors.min.js
33         ?ver=<?php echo $fasc_plugin_ver; ?>"></script>
34
35
36
37 eval_node: Found taint in echo
38  Line| Content
39     9| $fasc_plugin_ver = $_GET['ver'];
40    35| <link rel="stylesheet" href="jquery.minicolors.css
41         ?ver=<?php echo $fasc_plugin_ver; ?>">
42
43
44
45 eval_node: Found taint in echo
46  Line| Content
47     9| $fasc_plugin_ver = $_GET['ver'];
48    36| <link rel="stylesheet" href="popup.css
```

```
49          ?ver=<?php echo $fasc_plugin_ver; ?>">
50
51
52
53 eval_node: Found taint in echo
54  Line| Content
55    39| var ajax_url = "<?php echo $_GET['ajaxurl']; ?>";
56
57
58
59 eval_node: Found taint in echo
60  Line| Content
61     9| $fasc_plugin_ver = $_GET['ver'];
62    41| <script type="text/javascript" src="popup.min.js
63          ?ver=<?php echo $fasc_plugin_ver; ?>"></script>
64
65
66 Node count: 74
```

Additionally, a much shorter sample from the O'Reilly PHP learning resources[1] was observed to have an XSS vulnerability. On line 19 of the code sample in Listing 5.2 below it can be seen that the value of $_POST['my_name'] is printed directly onto the page.

Listing 5.2: Example 2

```
1  <?php
2  // Code from https://github.com/oreillymedia/Learning_PHP/
3  blob/master/code/forms/forms-122.php
4
5
6  // Logic to do the right thing based on
7  // the request method
8  if (\$_SERVER['REQUEST_METHOD'] == 'POST') {
9      // If validate_form( ) returns errors, pass them to ...
           show_form( )
10     if (\$form_errors = validate_form( )) {
11         show_form(\$form_errors);
12     } else {
13         process_form( );
14     }
15 } else {
16     show_form( );
17 }
18 // Do something when the form is submitted
```

[1] https://github.com/oreillymedia/Learning_PHP/blob/master/code/forms/forms-122.php

```
function process_form( ) {
    print "Hello, ". \$_POST['my_name'];
}
// Display the form
function show_form(\$errors = array()) {
    // If some errors were passed in, print them out
    if (\$errors) {
        print 'Please correct these errors: <ul><li>';
        print implode('</li><li>', \$errors);
        print '</li></ul>';
    }
    print<<<_HTML_
<form method="POST" action="\$_SERVER[PHP_SELF]">
Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
</form>
_HTML_;
}
// Check the form data
function validate_form( ) {
    // Start with an empty array of error messages
    \$errors = array( );
    // Add an error message if the name is too short
    if (strlen(\$_POST['my_name']) < 3) {
        \$errors[  ] = 'Your name must be at least 3 letters long.';
    }
    // Return the (possibly empty) array of error messages
    return \$errors;
}
```

5.2 Effectiveness

Of the samples studied, all known XSS vulnerabilities were detected. A number of samples were pulled from public code repositories including Wordpress plugins and PHP learning resources, while others were crafted manually to test core functionality of the system. Table 5.1 below demonstrates the some of the sources and sinks identified and the

Table 5.1: Taint Flow

File	Taint Source	Sensitive Sink
forms-122.php	`$_POST['my_name']`	`print $_POST['my_name']`
popup.php	`$fasc_plugin_ver = $_GET['ver']`	`echo $fast_plugin_ver;`
edit-styles.php	`$_GET['gid']`	`echo $_GET['gid'];`

files in which the taint flow was found.

5.3 Running Time

Each real-world sample was evaluated fifty times and the runtime was determined for each test using the GNU time command. The time was calculated by taking the sum of the user-mode time (User) and the time spent in system calls (Sys). The average of these fifty trials for each code sample was then calculated. Below is a graph of the average runtime for a sample against the number of nodes in the AST representation of that program.

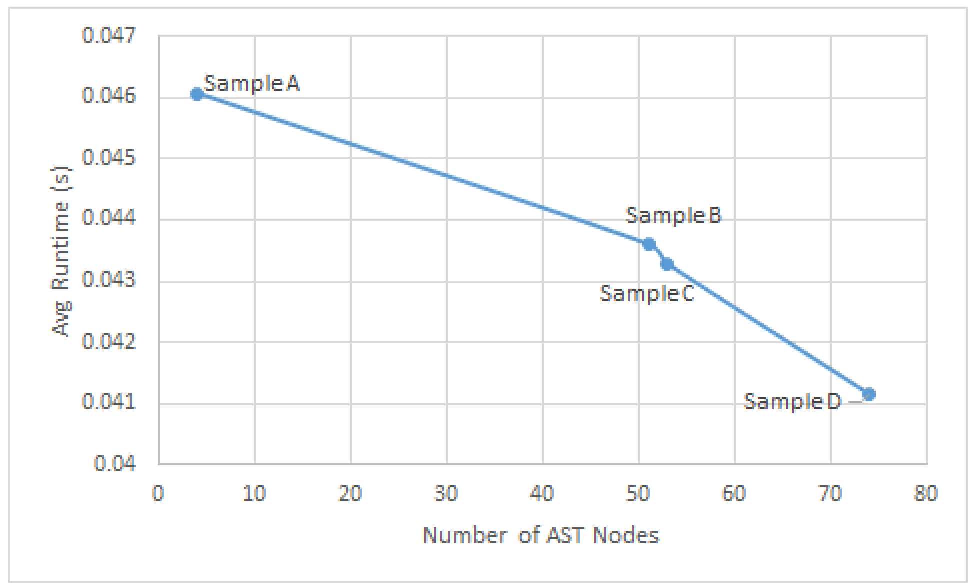

Figure 5.1: Effect of AST Node Count on Running Time

Discussion

We believe the program has been shown to be effective on the provided samples of PHP code and on the test cases designed to thoroughly test how the program handles analysis of specific elements of the PHP language. As was mentioned in the previous section a relatively small quantity of testable code was available for analysis due to the heavy reliance on OOP by modern PHP projects, handling for which was omitted from this project as it was left for future work. Implementing support for this feature of the language would require some reworking of the code to support classes, attributes and class functions, however it would be feasible to add to the existing codebase and is required to process more complicated code such as that from frameworks like Wordpress and Drupal (and their plugins).

Persistent XSS vulnerabilities are not supported in the program to as we specifically target reflected XSS vulnerabilities. Addition of this functionality would be simple, requiring identification of any read/write operations to a database management system (DBMS) so that the flow of data into and out of the database could be evaluated for taint. Whether or not data from the database is considered tainted would be dependent on the current state of the database and if data already present could be assumed to be correctly sanitized. This ”explicit trust” could be implemented as a flag in the program to allow users to specify whether or not read operations from the data source should be assumed to be safe, as users with existing databases may not be able to trust that all data stored there is not tainted.

Support for arrays currently assumes that square bracket notation will be utilized,

allowing for the tainting of specific indices within the array. Use of the array() construct would require some reworking of the array handling code. This arises due to the fact that an array assigned with the $array(key_1 \rightarrow value_1, ..., key_n \rightarrow value_n)$ notation has two methods of accessing values: by key, and by integer index. If keys are specified, the integer indices are automatically generated and assigned based on the order of key-value pair assignments. However, manual index specification can permit ”gaps” in the array, such as the following example in Listing 6.1 from the PHP documentation[20]. While it does not appear to be common practice to use this method of array creation among PHP developers, support for this would be feasible to implement.

Listing 6.1: PHP Array Gaps

```
<?php
$array = array(
         "a",
         "b",
    6 => "c",
         "d",
);
var_dump($array);
?>

Resulting array:
array(4) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [6]=>
  string(1) "c"
  [7]=>
  string(1) "d"
}
```

Functions defined within the code to be analyzed are currently evaluated effectively. Aside from specific PHP built-in functions which are known to remove script from a string, in this current implementation any other built-in functions are assumed to have a return and

return taint only if any arguments are tainted. PHP-Parser does not implicitly analyze built-in functions, so the PHP functions would need to be either emulated or the source code provided and analyzed to determine both whether the function passes taint and what other effects it might have.

Support for file includes was also excluded from the scope of this project. Addition of this feature would require changes to scoping as any statements executed in an included file would need to be reflected within the current environment, however this could be added to the creation of the AST and the code included inline and would be simple to implement.

Conclusion

This thesis has described a prototype system for performing static analysis on PHP code with the goal of identifying XSS vulnerabilities. We demonstrate the use of the system on examples that display the core functionality of the program, detailing both the implementation and challenges that exist in analyzing PHP code. A number of tasks exist for future work, including adding support for object-oriented programming, handling formatted output statements (for completeness), differentiating between local and global variables, and supporting the alternate array definition notation listed in the previous section. These changes would all be beneficial and increase the amount of code that the program could successfully analyze for vulnerabilities.

Bibliography

[1] 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. *A New Framework of Security Vulnerabilities Detection in PHP Web Application*, 2015.

[2] OWASP. Owasp top 10 2013 - the ten most critical web application security risks. Technical report, OWASP Foundation, 2013.

[3] David Scott and Richard Sharp. Abstracting application-level web security. In *WWW 2002*, 2002.

[4] Andy Ide. Php just grows & grows. Online, January 2013. https://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html.

[5] ASE/IEEE International Conference on Social Computing and ASE/IEEE International Conference on Privacy, Security, Risk and Trust. *Quality of WordPress Plug-Ins: An Overview of Security and User Ratings*, 2012.

[6] Wikipedia. Static program analysis, 2016. https://en.wikipedia.org/wiki/Static_program_analysis.

[7] Wikipedia. Dynamic program analysis, 2016. https://en.wikipedia.org/wiki/Dynamic_program_analysis.

[8] Wikipedia. Taint checking, 2016. https://en.wikipedia.org/wiki/Taint_checking.

[9] OWASP. Static code analysis, 07 2016. https://www.owasp.org/index.php/Static_Code_Analysis#Data

[10] Wikipedia. Cross-site scripting, 2016. https://en.wikipedia.org/wiki/Cross-site_scripting.

[11] Wikipedia. Same-origin policy, 2016. https://en.wikipedia.org/wiki/Same-origin_policy.

[12] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW 2004*, 2004.

[13] Nenad Jovanovic, Christopher Kreugel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, (18):861–907, 2010.

[14] 2012 Working Conference on Reverse Engineering. *Fast Detection of Access Control Vulnerabilities in PHP Applications*. IEEE Computer Society, 2012.

[15] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *NDSS '14*, 2014.

[16] Johannes Dahse. Rips - a static source code analyzer for vulnerabilities in php scripts, 2016.

[17] 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. *phpSAFE: A Security Analysis Tool for OOP Web Application Plugins*, 2015.

[18] Iberia Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, March 2016.

[19] *Vetting browser extensions for security vulnerabilities with VEX*, 2010.

[20] PHP. Assignment operators, 2016. http://php.net/manual/.

Appendix A

Program Code

Listing A.1: Program Code

```
#!/usr/bin/env php

<?php


    // Bootstrap generated by composer
    require 'vendor/autoload.php';

    // Increase debug level
    ini_set('xdebug.max_nesting_level', 3000);

    $debug = 0;
    $scope_debug = False;


    // Load the PhpParser library
    use PhpParser\Error;
    use PhpParser\ParserFactory;
    use PhpParser\PrettyPrinter;
    use PhpParser\NodeTraverser;
    use PhpParser\NodeVisitor\NameResolver;
    use PhpParser\NodeDumper;

    // Use our custom Node Visitor class
    require __DIR__ . '/node_visitor.php';

    // Array of tainted arrays (PHP "superglobals")
    $tainted_arrays = array("_GET",
                            "_POST",
                            "_FILES",
                            "_COOKIE",
                            "_REQUEST");

```

```
34      // Array of special functions that introduce or remove taint
35      $special_functions = array( 'htmlspecialchars',
36                                  'htmlentities');
37
38
39      /*
40          In PHP, printing a bool returns "1" if true and "" if false
41          This function fixes this PHP "feature" and returns the ...
                strings "True" or "False"
42      */
43      function print_bool($val){
44          if($val){
45              return "True";
46          } else {
47              return "False";
48          }
49      }
50
51
52      /*
53          Small function to return an X if the given value is true, ...
                used for printing the taint array.
54      */
55      function x_if_true($val){
56          if($val){
57              return "x";
58          }
59          return "";
60      }
61
62      /*
63          Function to pretty-print the $vars array, since print_r ...
                takes up too much space in the output
64      */
65      function print_vars($vars_arr = NULL){
66          global $vars;
67
68          if(is_null($vars_arr)){
69              $vars_arr = $vars;
70          }
71
72
73          $mask = "|%-20.20s|%-5s|%-50s\n";
74          printf($mask, "Variable", "Taint", "Taint Flow (Line ...
                Numbers)");
75          printf($mask, "--------------------", "-----", ...
                "------------------------------");
76
77          foreach($vars_arr as $key => $value){
78              $taint_flow = implode(', ', $value['taint_flow']);
79              printf($mask, $key, x_if_true($value['tainted']), ...
                    $taint_flow);
80          }
81
```

```
82              echo "\n";
83          }
84
85
86          /*
87              At the first sighting of a variable, initialize the data ...
                    structure for it in the $vars array
88          */
89          function init_var($var_name){
90              global $vars;
91              // echo "init_var: First usage of variable " . $var_name ...
                    . "\n";
92              $vars[$var_name] = array(
93                  'tainted' => False,
94                  'taint_flow' => array(),
95              );
96          }
97
98
99          /*
100             Return the taint status of a particular variable
101         */
102         function is_tainted($var_name){
103             global $vars;
104
105             if(array_key_exists($var_name, $vars)){
106                 return $vars[$var_name]['tainted'];
107             } else {
108                 return False;
109             }
110         }
111
112
113         /*
114             Given a variable name, mark it as tainted
115         */
116         function taint($var_name){
117             global $vars;
118             $vars[$var_name]['tainted'] = True;
119
120         }
121
122
123         /*
124             Given a variable name, mark it as untainted
125         */
126         function untaint($var_name){
127             global $debug;
128             global $vars;
129
130             if($debug){
131                 echo "untaint: Removing taint from variable $var_name\n";
132             }
133             $vars[$var_name]['tainted'] = False;
```

```
134     }
135
136
137     /*
138         Creates the key used in the vars[] array.
139         Specifically used to handle things like array indices, ...
            properties, etc.
140     */
141     function get_var_name($node){
142
143         global $debug;
144
145         switch(get_class($node)){
146             case "PhpParser\Node\Expr\Variable":
147                 return $node->name;
148
149             case "PhpParser\Node\Expr\ArrayDimFetch":
150                 $array_name = get_var_name($node->var);
151                 $key_name = get_var_name($node->dim);
152
153                 // Differentiate between arrays indexed by ...
                    strings and numbers
154                 if(is_a($node->dim, ...
                    "PhpParser\Node\Scalar\LNumber")){
155                     $var_name = $array_name . "[" . $key_name . "]";
156                 } elseif(is_a($node->dim, ...
                    "PhpParser\Node\Scalar\String_")){
157                     $var_name = $array_name . "['" . $key_name . ...
                        "']";
158                 }
159
160
161                 // $var_name = $array_name . "['" . $key_name . "']";
162                 // $var_name = addslashes($var_name);
163                 if($debug > 2){
164                     echo "get_var_name: " . $var_name . "\n";
165                 }
166                 return $var_name;
167
168             case "PhpParser\Node\Scalar\String_":
169                 return $node->value;
170
171             case "PhpParser\Node\Scalar\LNumber":
172                 return $node->value;
173         }
174     }
175
176
177     /*
178         Helper function to get the taint flow information when ...
            given a variable name. Can also be
179         used if you already have the variable name. Returns an ...
            array of line numbers.
180     */
```

```
181     function _get_taint_flow($var_name){
182         global $debug;
183         global $vars;
184         return $vars[$var_name]['taint_flow'];
185     }
186
187
188     /*
189         For a given AST, find the history of taint flow and ...
                return it as an array of line numbers
190     */
191     function get_taint_flow($node){
192         global $debug;
193         global $vars;
194         global $tainted_arrays;
195
196         if(is_null($node)){
197             return array();
198         }
199
200         $var_name = get_var_name($node);
201
202         $node_class = get_class($node);
203
204         switch($node_class){
205             case "PhpParser\Node\Expr\Variable":
206                 if(is_tainted($var_name)){
207                     return _get_taint_flow($var_name);
208                 } else {
209                     return array(); //return an empty array, ...
                            nothing to see here
210                 }
211
212
213             case "PhpParser\Node\Expr\ArrayDimFetch":
214                 // Get the names of the array, the dim, and ...
                        combined (in the notation array[dim])
215                 $array_name = get_var_name($node->var);
216                 $key_name = get_var_name($node->dim);
217                 $var_name = get_var_name($node);
218
219                 // Special case if it's a tainted array, just ...
                        return the number of this current line
220                 // Note that i.e. $_POST[$var] is technically two ...
                        sources of taint, but for our purposes we just ...
                        need to know that this $_POST generates taint
221
222                 if(in_array($array_name, $tainted_arrays)){
223                     return array($node->getLine());
224                 }
225
226                 // If it's just an array indexed with a scalar, ...
                        look up that specific key in the array
227                 if($node->dim instanceof PhpParser\Node\Scalar){
```

```
228                     return _get_taint_flow($var_name);
229                 // Otherwise, check the taint history of the ...
                    variable that serves as the array index too
230                 } else {
231                     return array_merge(taint_flow($var_name), ...
                        _get_taint_flow($key_name));
232                 }
233
234
235             case (preg_match("/BinaryOp.*/", $node_class) ? ...
                    $node_class : !$node_class):
236                 return array_merge(get_taint_flow($node->left), ...
                    get_taint_flow($node->right));
237
238
239             case "PhpParser\Node\Scalar\String_":
240                 return array();
241
242
243             case "PhpParser\Node\Scalar\LNumber":
244                 return array();
245
246
247             case "PhpParser\Node\Expr\ArrayItem":
248                 return array_merge(get_taint_flow($node->key), ...
                    get_taint_flow($node->value));
249
250
251             default:
252                 return array();
253         }
254     }
255
256
257
258     /*
259         Append taint flow history to a variable, given an array ...
            of integers corresponding to line numbers
260     */
261     function add_taint_flow($var_name, $history){
262         global $vars;
263         // Remove duplicate line number values, and assign to ...
            this var
264         $vars[$var_name]['taint_flow'] = ...
            array_unique($vars[$var_name]['taint_flow'] + $history);
265     }
266
267
268     /*
269         Variable $var_name is untainted, so clear the taint flow ...
            array
270     */
271     function clear_taint_flow($var_name){
272         global $vars;
```

```
273         $vars[$var_name]['taint_flow'] = array();
274     }
275 
276 
277 
278     /*
279         Merge two $vars arrays
280         PHP's array diff only works in one dimension, so we need ...
               to do this manually
281     */
282     function merge_vars_arrays($new_vars, $old_vars, $mandatory = ...
            False){
283         $result_arr = array();
284         // Check each var in the old array and see if it's in the ...
               new one. If the variable is in the new array and the ...
               old one, it needs to be updated. Anything not in the ...
               old array will be thrown out (end of scope).
285         foreach($old_vars as $key => $value){
286             if(array_key_exists($key, $new_vars)){
287                 $result_arr[$key] = ...
                        merge_vars_props($new_vars[$key], ...
                        $old_vars[$key], $mandatory);
288             } else {
289                 $result_arr[$key] = $value;
290             }
291         }
292 
293         return $result_arr;
294     }
295 
296 
297 
298     /*
299         When merging vars arrays, merge their properties in a ...
               smart way. I.e. if either variable is
300         tainted, the result is tainted. Additionally, merge their ...
               taint flow histories.
301 
302         NOTE:
303         Each variable in the $vars array is associated with a ...
               number of properties. We need to find
304         any differences between the old $vars array and the new ...
               $vars array and
305     */
306     function merge_vars_props($old_props, $new_props, $mandatory ...
            = False){
307         // Compare the properties array for this variable name ...
               between the old and new array. NOTE: Err on the side ...
               of caution; if it lost taint, it might still have it ...
               in another branch, so don't remove it
308         $ret_props = array();
309         // First handle the tainted property; if either was ...
               tainted, the result should still
310 
```

```
311         if(!$mandatory){
312             $ret_props['tainted'] = $old_props['tainted'] || ...
                    $new_props['tainted'];
313         } else {
314             if($new_props['tainted'] == False){
315                 // If it was a mandatory conditional, and the ...
                        variable is still false, that means that every ...
                        branch resulted in the variable being ...
                        untainted; therefore, we can untaint it in the ...
                        global $vars array and remove the taint history.
316                 $ret_props['tainted'] = False;
317                 $old_props['taint_flow'] = array();
318                 $new_props['taint_flow'] = array();
319             } else {
320                 // Otherwise, it must be tainted so set it to True
321                 $ret_props['tainted'] = True;
322             }
323         }
324
325         // Now merge the taint_flow arrays
326         $ret_props['taint_flow'] = ...
                array_unique(array_merge($old_props['taint_flow'], ...
                $new_props['taint_flow']));
327
328         return $ret_props;
329     }
330
331
332     /*
333         For special PHP functions, emulate their behavior here. ...
                This covers functions
334         that always generate of remove taint.
335     */
336     function handle_function($func_name, $args){
337         switch($func_name){
338
339             case "htmlspecialchars":
340                 return False;
341
342             case "htmlentities":
343                 return False;
344
345             default:
346                 echo "Handling function $func_name\n";
347                 // We'll assume it doesn't remove taint
348                 foreach($args as $arg){
349                     if(eval_node($arg)){
350                         return True;
351                     }
352                 }
353                 return False;
354         }
355
356     }
```

```
357
358
359     /*
360         Recursively evaluates nodes from the AST to determine ...
                their taint status
361
362         Requires: A node of type PhpParser\Node\*
363         Returns: True or false for expressions (indicating their ...
                taint status), otherwise nothing
364     */
365     function eval_node($node){
366
367         // Tell PHP to let us use our global in this scope
368         // What a great language
369         global $debug;
370         global $nodeDumper;
371         global $prettyPrinter;
372
373         global $vars;
374         global $functions;
375         global $special_functions;
376         global $lines;
377         global $has_return;
378         global $returns_taint;
379         global $returns_taint_history;
380         global $break_hit;
381         global $tainted_arrays;
382
383         // print_r($node);
384
385         $currentLine = $node->getLine();
386
387         $node_class = get_class($node);
388
389         echo "eval_node: Evaluating " . $node_class . "\n";
390
391         switch($node_class){
392
393             case "PhpParser\Node\Expr\Assign":
394                 // Get the name of the variable to which we're ...
                        assigning
395                 $var_name = '';
396                 if(!($node->var instanceof ...
                        PhpParser\Node\Expr\Variable)){
397                     $var_name = get_var_name($node->var);
398                 } else {
399                     $var_name = $node->var->name;
400                 }
401
402                 // If it's not in the array, initialize the ...
                        variable being assigned to as untainted and ...
                        create an empty array that will track line ...
                        numbers to show taint flow
403                 if(!array_key_exists($var_name, $vars)){
```

```
404                     init_var($var_name);
405                 }
406 
407                 // If the expression being assigned has taint, ...
                        find the flow and add that to the assignee
408                 if(eval_node($node->expr)){
409                     echo "eval_node: Taint propagated on line ...
                            number " . $currentLine . "\n";
410 
411                     // Taint the variable
412                     taint($var_name);
413 
414                     // Get the taint history of the item(s) being ...
                            assigned
415                     if(is_a($node->expr, ...
                            "PhpParser\Node\Expr\FuncCall")){
416                         $history = $returns_taint_history;
417                     } else {
418                         $history = get_taint_flow($node->expr);
419                     }
420 
421                     // Add the taint flow history plus the ...
                            current line num to this variable's flow ...
                            history
422                     $flow_to_add = array_merge($history, ...
                            array($currentLine));
423 
424                     add_taint_flow($var_name, $flow_to_add);
425 
426                 } else {
427                     // Remove taint and clear taint_flow array
428                     untaint($var_name);
429                     clear_taint_flow($var_name);
430                 }
431 
432                 return;
433 
434 
435             // Echo statements are sinks for taint in XSS
436             case "PhpParser\Node\Stmt\Echo_":
437                 foreach($node->exprs as $expr){
438                     if(eval_node($expr)){
439                         echo "\neval_node: Found taint in echo!\n";
440 
441                         $history = get_taint_flow($expr);
442                         $history[] = $currentLine;
443 
444                         printf("%5s| %s", "Line", "Content\n");
445 
446                         foreach($history as $line){
447                             printf("%5d| %s\n", $line, ...
                                    trim($lines[$line-1]));
448                         }
449 
```

```
450                         echo "\n\n";
451                         return;
452                     }
453                 }
454                 return;
455 
456             // Print statements are sinks for taint in XSS
457             case "PhpParser\Node\Expr\Print_":
458                 if(eval_node($node->expr)){
459                     echo "\neval_node: Found taint in print!\n";
460 
461                     $history = get_taint_flow($node->expr);
462                     $history[] = $currentLine;
463 
464                     printf("%5s| %s", "Line", "Content\n");
465                     foreach($history as $line){
466                         printf("%5d| %s\n", $line, ...
                                trim($lines[$line-1]));
467                     }
468                     echo "\n\n";
469                 }
470                 return;
471 
472 
473             case "PhpParser\Node\Stmt\Do_":
474                 // Evaluate the while first, then run through our ...
                        loop
475                 $saved_vars = $vars;
476 
477                 eval_node($node->cond);
478 
479                 foreach($node->stmts as $stmt){
480                     eval_node($stmt);
481                 }
482 
483                 // Merge any updates back in to the main array, ...
                        remove any locals
484                 $vars = merge_vars_arrays($vars, $saved_vars);
485 
486                 break;
487 
488             case "PhpParser\Node\Stmt\For_":
489                 $saved_vars = $vars;
490 
491                 // Evaluate the initial condition(s)
492                 foreach($node->init as $stmt){
493                     eval_node($stmt);
494                 }
495 
496                 // Check the loop action(s)
497                 foreach($node->loop as $stmt){
498                     eval_node($stmt);
499                 }
500 
```

```
501                 // Evaluate the statements inside the loop
502                 foreach($node->stmts as $stmt){
503                     eval_node($stmt);
504                 }
505 
506                 $vars = merge_vars_arrays($vars, $saved_vars);
507 
508                 return;
509 
510 
511             case "PhpParser\Node\Stmt\Foreach_":
512                 $saved_vars = $vars;
513 
514                 // Find if any items in the target array are tainted
515                 $tainted = False;
516                 $flow = array();
517                 foreach($vars as $var => $values){
518                     if(preg_match('/^' . ...
                            get_var_name($node->expr) . '/', $var) === 1){
519                         if(is_tainted($var)){
520                             $tainted = True;
521 
522                             // Now get the taint flow so we can ...
                                    see where the source was
523                             $flow = ...
                                    array_unique(_get_taint_flow($var) ...
                                    + $flow);
524                         }
525                     }
526                 }
527 
528                 // Add our loop variable and set the taint status ...
                        if needed; merge taint status if it exists
529                 init_var(get_var_name($node->valueVar));
530                 if($tainted){
531                     taint(get_var_name($node->valueVar));
532                 }
533 
534                 add_taint_flow(get_var_name($node->valueVar), $flow);
535 
536                 foreach($node->stmts as $stmt){
537                     eval_node($stmt);
538                 }
539 
540                 // Update any changed vars, remove locally scoped ...
                        vars
541                 $vars = merge_vars_arrays($vars, $saved_vars);
542 
543                 return;
544 
545 
546             case "PhpParser\Node\Stmt\If_":
547                 // Save the original vars so we know what's changed
548                 $saved_vars = $vars;
```

```
549
550                     // Here's where we'll merge the results of taint ...
                           from each block
551                     $merged_vars = array();
552
553                     // Evaluate the initial if condition statements
554                     foreach($node->stmts as $stmt){
555                         eval_node($stmt);
556                     }
557
558                     // Merge an empty array with our current vars ...
                           array at the end of the If branch
559                     $merged_vars = merge_vars_arrays($merged_vars, ...
                           $vars);
560
561                     if($debug){
562                         echo "eval_node: Results from Stmt_If:\n";
563                         print_vars();
564                     }
565
566                     // Evaluate each elseif and merge in the taint ...
                           status and history
567                     if(!is_null($node->elseifs)){
568                         foreach($node->elseifs as $elseif){
569                             $vars = $saved_vars;
570                             eval_node($elseif);
571
572                             // Merge the results from the If branch ...
                                   with this ElseIf branch
573                             echo "eval_node: Merging result from ...
                                   ElseIf on line $currentLine\n";
574                             $merged_vars = ...
                                   merge_vars_arrays($merged_vars, $vars);
575                         }
576                     }
577
578                     // Evaluate the else and merge in the taint ...
                           status and history
579                     if(!is_null($node->else)){
580                         $vars = $saved_vars;
581                         eval_node($node->else);
582
583                         // Merge the results from the Else branch ...
                               with all previous results
584                         $merged_vars = ...
                               merge_vars_arrays($merged_vars, $vars);
585                     }
586
587                     // NOTE: Now we can update our original vars ...
                           array. If the statement is of the form ...
                           if->else, and all branches remove taint from a ...
                           particular variable, then it is guaranteed ...
                           that the taint will be removed and we pass a ...
                           True which signals to the function that this ...
```

```
                    is a "mandatory" conditional and at least one ...
                    of these must be executed. The "mandatory" ...
                    flag allows the removal of taint; conditionals ...
                    with only an If or If/ElseIf are not ...
                    guaranteed to execute therefore we cannot ...
                    assume that a "False" tainted result for a ...
                    variable will occur in every case.
588
589                 if(!is_null($node->else)){
590                     $vars = merge_vars_arrays($saved_vars, ...
                            $merged_vars, True);
591                     if($debug){
592                         echo "eval_node: If: Results from ...
                                mandatory merge:\n";
593                         print_vars();
594                     }
595                 } else {
596                     $vars = merge_vars_arrays($saved_vars, ...
                            $merged_vars);
597                     if($debug){
598                         echo "eval_node: If: Results from ...
                                non-mandatory merge:\n";
599                         print_vars();
600                     }
601                 }
602
603                 return;
604
605
606             case "PhpParser\Node\Stmt\ElseIf_":
607                 foreach($node->stmts as $stmt){
608                     eval_node($stmt);
609                 }
610
611                 if($debug){
612                     echo "eval_node: Results from Stmt_ElseIf\n";
613                     print_vars();
614                 }
615                 return;
616
617
618             case "PhpParser\Node\Stmt\Else_":
619                 foreach($node->stmts as $stmt){
620                     eval_node($stmt);
621                 }
622                 if($debug){
623                     echo "eval_node: Results from Stmt_Else\n";
624                     print_vars();
625                 }
626                 return;
627
628             case "PhpParser\Node\Stmt\Function_":
629                 // Shouldn't have any redeclarations, because ...
                    it'd throw
```

```
630                 // a fatal PHP error, but let's just be safe
631                 if(!array_key_exists($node->name, $functions)){
632                     $functions[$node->name] = $node;
633                     echo "eval_node: Adding function $node->name ...
                            to function list\n";
634                 }
635 
636                 return;
637 
638 
639             case "PhpParser\Node\Expr\FuncCall":
640                 // Reset our return flags
641                 $has_return = False;
642                 $returns_taint = False;
643 
644                 // Get the name of the function we're calling
645                 $func_name = $node->name->parts[0];
646 
647                 // Gather args
648                 $args = $node->args;
649 
650                 // If it's a special, escaping function, go ...
                        handle it
651                 // Alternatively, if function_exists returns ...
                        True, it's a built-in
652                 if(in_array($func_name, $special_functions) || ...
                        function_exists($func_name)){
653                     // Assume it has a return, since parser ...
                            doesn't know for built-ins
654                     $has_return = True;
655                     $returns_taint = handle_function($func_name, ...
                            $args);
656                 } else {
657 
658 
659                     // Save our current variables and remove all ...
                            existing vars
660                     // (since we're in a function scope now)
661                     $saved_vars = $vars;
662 
663                     // Create a safe place to store ...
                            function-local vars because
664                     // we'll need to wipe the vars array to ...
                            emulate function scope
665                     $tmp_vars = array();
666 
667                     // Make sure this is a function we've seen ...
                            defined; if not, print a warning and skip
668                     if(!array_key_exists($func_name, $functions)){
669                         echo "eval_node: NOTICE: Function not ...
                                found in defined list: $func_name\n";
670                         print_r($functions);
671                         return;
672                     }
```

```
673
674
675                     // Now, define the function's parameters ...
                           based on the args
676                     $params = $functions[$func_name]->params;
677
678
679                     // Iterate through each argument
680                     for($i=0; $i<count($args); $i++){
681                         // Get name of param
682                         $var_name = $params[$i]->name;
683                         $tmp_vars[$var_name] = array(
684                             'tainted' => eval_node($args[$i]->value),
685                             'taint_flow' => ...
                                   get_taint_flow($args[$i]->value),
686                         );
687
688                         // If it was tainted coming in as an arg, ...
                               add the line of the function call to ...
                               the history
689                         if($tmp_vars[$var_name]['tainted']){
690                             array_push($tmp_vars[$var_name]['taint_flow'], ...
                                   $currentLine);
691                         }
692
693                     }
694
695                     // Leaves only the function local variables
696                     $vars = $tmp_vars;
697                     if($debug){
698                         echo "Scope inside function $func_name:\n";
699                         print_vars($vars);
700                     }
701
702
703                     // TODO: Handle return values! Add handler ...
                           for Stmt\Return_ class.
704                     $stmts = $functions[$func_name]->stmts;
705
706                     // Emulate each statement in the function
707                     foreach($stmts as $stmt){
708                         eval_node($stmt);
709                     }
710
711
712                     // Merge any updates back in to the main ...
                           array, remove any locals
713                     $vars = merge_vars_arrays($vars, $saved_vars);
714                 }
715
716                 if($has_return){
717                     return $returns_taint;
718                 } else {
719                     return;
```

```
720                 }
721
722
723         case "PhpParser\Node\Stmt\Return_":
724             $tainted = eval_node($node->expr);
725
726             // Log that this function returns data
727             $has_return = True;
728
729             if($tainted){
730                 // Add the history of the taint sources in ...
                        this return
731                 $returns_taint_history = ...
                        array_merge($returns_taint_history, ...
                        get_taint_flow($node->expr));
732
733                 // Also add this return statement so the ...
                        whole path can be seen
734                 $returns_taint_history = ...
                        array_merge($returns_taint_history, ...
                        array($currentLine));
735             }
736
737             // Helps track if any returns generate taint
738             $returns_taint = $returns_taint || $tainted;
739
740             return $tainted;
741
742
743         case "PhpParser\Node\Arg":
744             return eval_node($node->value);
745
746
747         case "PhpParser\Node\Stmt\Switch_":
748             $saved_vars = $vars;
749             $merged_vars = array();
750
751             $cases = $node->cases;
752
753             // NOTE: If there's a default, and the end result ...
                    has a variable untainted, then that variable ...
                    has actually been guaranteed to be untainted
754             $is_default = False;
755
756             // Evaluate each case statement sequentially.
757
758             for($i = 0; $i < count($cases); $i++){
759                 // echo "Starting on case $i:\n";
760
761                 // A null condition means this is the ...
                        "default" case
762                 if(is_null($cases[$i]->cond)){
763                      $is_default = True;
764                 }
```

```
765
766                     // print_r($cases[$i]);
767                     for($j = $i; $j < count($cases) && ...
                            !$break_hit; $j++){
768
769                         // Evaluate the statements for this case ...
                                statement
770                         foreach($cases[$j]->stmts as $stmt){
771                             eval_node($stmt);
772                         }
773                     }
774
775
776                     // Merge the results of this branch
777                     $merged_vars = ...
                            merge_vars_arrays($merged_vars, $vars);
778
779                     // Reset the vars array
780                     $vars = $saved_vars;
781
782                     // Reset the break flag
783                     $break_hit = False;
784
785                 }
786
787                 if($is_default){
788                     $vars = merge_vars_arrays($saved_vars, ...
                            $merged_vars, True);
789                     if($debug){
790                         echo "eval_node: Switch: Results from ...
                                mandatory merge:\n";
791                         print_vars();
792                     }
793                 } else {
794                     $vars = merge_vars_arrays($saved_vars, ...
                            $merged_vars);
795                     if($debug){
796                         echo "eval_node: Switch: Results from ...
                                non-mandatory merge:\n";
797                         print_vars();
798                     }
799                 }
800
801                 return;
802
803
804             case "PhpParser\Node\Stmt\Break_":
805                 $break_hit = True;
806                 return;
807
808
809             case "PhpParser\Node\Stmt\While_":
810                 $saved_vars = $vars;
811
```

```
812                     eval_node($node->cond);
813
814                     foreach($node->stmts as $stmt){
815                         eval_node($stmt);
816                     }
817
818                     // Merge any updates back in to the main array, ...
                            remove any locals
819                     $vars = merge_vars_arrays($vars, $saved_vars);
820
821                     return;
822
823
824                 // Use regex since all BinaryOps are the same
825                 case (preg_match("/BinaryOp/", $node_class) ? ...
                        $node_class : !$node_class):
826                     if(eval_node($node->left) or ...
                            eval_node($node->right)){
827                         return True;
828                     }
829                     return False;
830
831
832                 case "PhpParser\Node\Expr\ArrayDimFetch":
833                     // If our key is the result of a tainted lookup, ...
                            this is now tainted
834                     $dim_tainted = eval_node($node->dim);
835                     $var_name = get_var_name($node);
836
837    $tainted_arrays)) . "\n";
838
839                     if($dim_tainted || in_array($node->var->name, ...
                            $tainted_arrays) || is_tainted($var_name)){
840                         if($debug){
841                             echo "eval_node: Taint generated in ...
                                    ArrayDimFetch on line $currentLine\n";
842                         }
843                         return True;
844                     } else {
845                         return False;
846                     }
847
848
849                 case "PhpParser\Node\Expr\ConstFetch":
850                     return False;
851
852
853                 case "PhpParser\Node\Expr\Variable":
854                     return is_tainted($node->name);
855
856
857                 case "PhpParser\Node\Scalar\Encapsed":
858                     $tainted = False;
859                     foreach($node->parts as $part){
```

```
860                         if(eval_node($part)){
861                             $tainted = True;
862                         }
863                     }
864 
865                     return $tainted;
866 
867 
868                 case "PhpParser\Node\Scalar\String_":
869                     return False;
870 
871 
872                 case "PhpParser\Node\Stmt\InlineHTML":
873                     return False;
874 
875 
876                 default:
877                     return False;
878         }
879     }
880 
881 
882 
883 
884 
885     // Track variable taint
886     $vars = array();
887 
888     // Store functions
889     $functions = array();
890 
891     // Track if a function returns any data
892     $has_return = False;
893 
894     // Flag for function taint return
895     $returns_taint = False;
896 
897     // Track return taint history
898     $returns_taint_history = array();
899 
900     // Flag to track when a switch-case has hit a break
901     $break_hit = False;
902 
903 
904     // Set up components
905     $parser = (new ...
            ParserFactory)->create(ParserFactory::PREFER_PHP5);
906     $traverser = new NodeTraverser;
907     $prettyPrinter = new PrettyPrinter\Standard;
908     $nodeDumper = new PhpParser\NodeDumper;
909 
910     if (isset($argv[1])){
911         $targetFile = $argv[1];
912     } else {
```

```
913         die("No file specified!\n");
914     }
915
916
917     // Content of file being analyzed
918     $lines = file($targetFile);
919
920
921     // Add our custom Name Resolver and Node Visitor to the traverser
922     $traverser->addVisitor(new NameResolver);
923     $traverser->addVisitor(new NodeVisitor);
924
925
926
927     // Open the file and parse the statements
928     try {
929         $file_contents = file_get_contents($targetFile);
930
931         // Print the length of the file
932         if ( $debug ) {
933             echo "Code is ", substr_count($file_contents, ...
                    PHP_EOL), " lines long.\n";
934         }
935
936         // Parse the file to retrieve an array of statements
937         $stmts = $parser->parse($file_contents);
938
939     } catch (Error $e) {
940         echo 'Parse Error: ', $e->getMessage();
941     }
942
943
944     // Traverse the nodes and convert to SSA form
945     echo "\n\nAbstract Syntax Tree:\n";
946     echo "---------------------\n";
947     echo $nodeDumper->dump($stmts) . "\n\n\n";
948
949
950
951
952
953     // Iterate over top level nodes; eval_node will recurse into ...
           the lower-level nodes automatically
954     foreach($stmts as $node){
955         eval_node($node);
956     }
957
958     echo "\n\n---------------------\n";
959     echo "Vars array at the end of analysis\n";
960     echo "---------------------\n";
961     print_vars();
962
963
964 ?>
```

Appendix B

Sample Code

Listing B.1: Example 1

```
1  <?php
2  // Code from ...
       https://wordpress.org/plugins/forget-about-shortcode-buttons/
3  // this file contains the contents of the popup window
4      $source = "insert";
5      $title = "Insert Button";
6      $insert_text = "Insert";
7      if(isset($_GET['ver']))
8      {
9          $fasc_plugin_ver = $_GET['ver'];
10     }
11     else
12     {
13         $fasc_plugin_ver = "";
14     }
15
16
17     if(isset($_GET['source']))
18     {
19         if($_GET['source']=="click")
20         {
21             $source = "click";
22             $title = "Edit Button";
23             $insert_text = "Update";
24         }
25     }
26 ?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" ...
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
27 <html xmlns="http://www.w3.org/1999/xhtml">
28 <head>
29 <title><?php echo $title; ?></title>
```

```
30 <script type="text/javascript" ...
       src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
31 </script>
32 <script language="javascript" type="text/javascript" ...
       src="../../../../../../wp-includes/js/tinymce/tiny_mce_popup.js
33 ?ver=<?php echo $fasc_plugin_ver; ?>"></script>
34 <link rel="stylesheet" ...
       href="../../css/button-styles.css?ver=<?php echo ...
       $fasc_plugin_ver; ?>" />
35 <link rel="stylesheet" href="../../css/font-awesome.css?ver=<?php ...
       echo $fasc_plugin_ver; ?>" />
36 <script src="jquery.minicolors.min.js?ver=<?php echo ...
       $fasc_plugin_ver; ?>"></script>
37 <link rel="stylesheet" href="jquery.minicolors.css?ver=<?php echo ...
       $fasc_plugin_ver; ?>">
38 <link rel="stylesheet" href="popup.css?ver=<?php echo ...
       $fasc_plugin_ver; ?>">
39 <script type="text/javascript">
40 var source = "<?php echo $source; ?>";
41 var ajax_url = "<?php echo $_GET['ajaxurl']; ?>";
42 </script>
43 <script type="text/javascript" src="popup.min.js?ver=<?php echo ...
       $fasc_plugin_ver; ?>"></script>
44 </head>
45 <body>
46
47     <!-- http://www.problogdesign.com/wordpress/
48 user-friendly-short-codes-with-tinymce/ -->
49     <!-- ...
           http://www.tinymce.com/develop/bugtracker_view.php?id=5575 -->
50
51     <div id="button-dialog">
52         <div class="preview-button-area" style="position:relative;">
53             <div class="centered-button">
54
55             </div>
56             <a href="#" class="fa fa-save save-btn"></a>
57         </div>
58
59         <form action="/" method="get" accept-charset="utf-8">
60
61             <div id="tab-header">
62                 <ul>
63                     <li class="active"><a ...
                           href="#tab-1-content">Properties</a></li>
64                     <li><a href="#tab-2-content">Icon</a></li>
65                     <li><a href="#tab-3-content">Templates</a></li>
66                     <!--<li class="settings"><a ...
                           href="#tab-4-content"><div ...
                           data-code="f111" class="dashicons ...
                           dashicons-admin-generic ...
                           active"></div></a></li>-->
67                 </ul><div class="clear"></div>
68             </div>
```

```
69
70              <div id="tab-1-content" class="fasc-tab-content">
71                  <div class="inputrow main">
72                      <label for="button-text">Text</label>
73                      <div class="inputwrap">
74                          <input type="text" name="button-text" ...
                                value="" id="button-text" value="" ...
                                placeholder="Enter your text&hellip;" />
75                      </div>
76                      <div class="clear"></div>
77                  </div>
78                  <div class="inputrow main">
79                      <label for="button-url">URL</label>
80                      <div class="inputwrap">
81                          <input type="text" name="button-url" ...
                                value="" id="button-url" /><br />
82                      </div>
83                      <div class="inputwrap">
84                          <label for="new-window"><small>Open link ...
                                in new window?</small></label> <input ...
                                type="checkbox" id="new-window" ...
                                name="new-window" value="1" />
85                      </div>
86                      <div class="clear"></div>
87                  </div>
88
89                  <div class="inputrow">
90
91                      <div class="inputcol left">
92                          <div class="inputrowc">
93                              <label for="text-color">Text ...
                                    Color</label>
94                              <div class="inputwrap">
95                                  <input type="text" ...
                                        id="text-color" ...
                                        name="text-color" ...
                                        value="#ffffff" />
96                              </div>
97                              <div class="clear"></div>
98                          </div>
99                          <div class="inputrowc">
100                             <label for="button-type">Type</label>
101                             <div class="inputwrap">
102                                 <select name="button-type" ...
                                        id="button-type">
103                                     <option ...
                                            value="fasc-type-flat" ...
                                            selected="selected">Flat ...
                                            </option>
104                                     <option value="fasc-type-flat ...
                                            fasc-rounded-medium">Flat ...
                                            Rounded</option>
105                                     <option ...
                                            value="fasc-type-glossy">Glossy ...
```

```
                                   </option>
106                             <option ...
                                   value="fasc-type-glossy ...
                                   fasc-rounded-medium">Glossy ...
                                   Rounded</option>
107                             <option ...
                                   value="fasc-type-popout">Pop ...
                                   out</option>
108                             <option ...
                                   value="fasc-type-popout ...
                                   fasc-rounded-medium">Pop ...
                                   out Rounded</option>
109                         </select>
110                     </div>
111                     <div class="clear"></div>
112                 </div>
113             </div>
114             <div class="inputcol right">
115 
116                 <div class="inputrowc">
117                     <label for="button-color">Button ...
                           Color</label>
118                     <div class="inputwrap">
119                         <input type="text" ...
                               id="button-color" ...
                               name="button-color" ...
                               value="#33809e" />
120                     </div>
121                     <div class="clear"></div>
122                 </div>
123                 <div class="inputrowc">
124                     <label for="button-size">Size</label>
125                     <div class="inputwrap">
126                         <select name="button-size" ...
                               id="button-size" size="1">
127                             <option ...
                                   value="fasc-size-xsmall"> ...
                                   Extra Small</option>
128                             <option ...
                                   value="fasc-size-small"> ...
                                   Small</option>
129                             <option ...
                                   value="fasc-size-medium" ...
                                   selected="selected"> ...
                                   Medium</option>
130                             <option ...
                                   value="fasc-size-large"> ...
                                   Large</option>
131                             <option ...
                                   value="fasc-size-xlarge"> ...
                                   Extra Large</option>
132                         </select>
133                     </div>
134                     <div class="clear"></div>
```

```
135                         </div>
136                     </div>
137 
138                     <div class="clear"></div>
139                 </div>
140 
141                 <div class="inputrow" style="display:none;">
142                     <label for="button-align">Alignment</label>
143                     <div class="inputwrap">
144                         <select name="button-align" ...
                                id="button-align" size="1">
145                             <option value="" ...
                                    selected="selected">None</option>
146                             <option value="left">Left</option>
147                             <option value="right">Right</option>
148                         </select>
149                     </div>
150                     <div class="clear"></div>
151                 </div>
152             </div>
153 
154             <div id="tab-2-content" class="fasc-tab-content">
155                 <div class="inputrow">
156                     <div class="inputwrap left">
157                         <select id="icon-type-select">
158                             <option ...
                                    value="dashicons-grid">Dashicons ...
                                    </option>
159                             <option value="fa-web-grid">Web </option>
160                             <option value="fa-media-grid">Media ...
                                    </option>
161                             <option value="fa-form-grid">Form ...
                                    </option>
162                             <option ...
                                    value="fa-currency-grid">Currency ...
                                    </option>
163                             <option value="fa-editor-grid">Editor ...
                                    </option>
164                             <option value="fa-directional-grid"> ...
                                    Directional </option>
165                             <option value="fa-brand-grid">Brand ...
                                    </option>
166                             <option ...
                                    value="fa-medical-grid">Medical ...
                                    </option>
167                         </select>
168                         </label>
169                     </div>
170 
171                     <div class="inputwrap right">
172 
173                         <label><input type="radio" ...
                                name="fasc-ico-position" value="none" ...
                                class="fasc-ico-position" ...
```

```
                                checked="checked" /> None</label>
174                          <label><input type="radio" ...
                                name="fasc-ico-position" ...
                                value="before" ...
                                class="fasc-ico-position" /> ...
                                Before</label>
175                          <!--<label><input type="radio" ...
                                name="fasc-ico-position" value="after" ...
                                /> After</label>
176                          <label><input type="radio" ...
                                name="fasc-ico-position" ...
                                value="center" /> Center</label>-->
177
178                      </div>
179                      <div class="clear"></div>
180                  </div>
181
182                  <div class="ico-grid">
183                      <div class="grid-container" id="dashicons-grid">
184                          <input type="hidden" value="" ...
                                id="fasc-ico-val" name="fasc-ico-val" />
185                          <?php require_once("dashicons-grid.php"); ?>
186                          <div class="clear"></div>
187                      </div>
188                      <div class="grid-container fontawesome" ...
                            id="fa-media-grid">
189                          <input type="hidden" value="" ...
                                id="fasc-ico-val" name="fasc-ico-val" />
190                          <?php require_once("fa-media-grid.php"); ?>
191                          <div class="clear"></div>
192                      </div>
193                      <div class="grid-container fontawesome" ...
                            id="fa-form-grid">
194                          <input type="hidden" value="" ...
                                id="fasc-ico-val" name="fasc-ico-val" />
195                          <?php require_once("fa-form-grid.php"); ?>
196                          <div class="clear"></div>
197                      </div>
198                      <div class="grid-container fontawesome" ...
                            id="fa-web-grid">
199                          <input type="hidden" value="" ...
                                id="fasc-ico-val" name="fasc-ico-val" />
200                          <?php require_once("fa-web-grid.php"); ?>
201                          <div class="clear"></div>
202                      </div>
203                      <div class="grid-container fontawesome" ...
                            id="fa-currency-grid">
204                          <input type="hidden" value="" ...
                                id="fasc-ico-val" name="fasc-ico-val" />
205                          <?php ...
                                require_once("fa-currency-grid.php"); ?>
206                          <div class="clear"></div>
207                      </div>
```

```
208                         <div class="grid-container fontawesome" ...
                                id="fa-editor-grid">
209                             <input type="hidden" value="" ...
                                    id="fasc-ico-val" name="fasc-ico-val" />
210                             <?php require_once("fa-editor-grid.php"); ?>
211                             <div class="clear"></div>
212                         </div>
213                         <div class="grid-container fontawesome" ...
                                id="fa-directional-grid">
214                             <input type="hidden" value="" ...
                                    id="fasc-ico-val" name="fasc-ico-val" />
215                             <?php ...
                                    require_once("fa-directional-grid.php"); ...
                                    ?>
216                             <div class="clear"></div>
217                         </div>
218                         <div class="grid-container fontawesome" ...
                                id="fa-brand-grid">
219                             <input type="hidden" value="" ...
                                    id="fasc-ico-val" name="fasc-ico-val" />
220                             <?php require_once("fa-brand-grid.php"); ?>
221                             <div class="clear"></div>
222                         </div>
223                         <div class="grid-container fontawesome" ...
                                id="fa-medical-grid">
224                             <input type="hidden" value="" ...
                                    id="fasc-ico-val" name="fasc-ico-val" />
225                             <?php ...
                                    require_once("fa-medical-grid.php"); ?>
226                             <div class="clear"></div>
227                         </div>
228                         <div class="clear"></div>
229                     </div>
230                 </div>
231                 <div id="tab-3-content" class="fasc-tab-content ...
                        saved-buttons-tab">
232
233                     <div class="container-grid">
234
235                         <ul>
236                         </ul>
237
238                         <div class="clear"></div>
239                     </div>
240                 </div>
241                 <div id="tab-4-content" class="fasc-tab-content">
242                     <strong>Button Attributes</strong>
243                     <div class="container-grid">
244
245
246
247                         <div class="clear"></div>
248                     </div>
249                 </div>
```

```
250                 <div id="fasc-footer">
251                     <a href="javascript:ButtonDialog.insert( ...
                            ButtonDialog.local_ed)" id="insert" ...
                            style="display: block; line-height: ...
                            24px;"><?php echo $insert_text; ?></a>
252                 </div>
253 
254 
255             </form>
256         </div>
257 
258 </body>
259 </html>
```