Static Detection of Complex Vulnerabilities in Modern PHP Applications

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

vorgelegt von

Johannes Dahse
aus Jena

Bochum, 02.02.2016

Gutachter:	Prof. Dr. Thorsten Holz (Ruhr-Universität Bochum)
Zweitgutachter:	Prof. Dr. Andrei Sabelfeld (Chalmers University of Technology)
Tag der mündlichen Prüfung:	11. März 2016

Abstract

Modern websites evolved to interactive applications which process confidential user data, such as credit card numbers, passwords, and private messages, on a daily basis. This sensitive data requires reliable protection from cyber criminals who exploit vulnerabilities in the applications' source code. Particularly web applications developed in PHP, the most popular server-side scripting language on the Web, are prone to security vulnerabilities. Although the developers' awareness is rising for the traditional types of vulnerabilities, such as *cross-site scripting* and *SQL injection*, they still persist due to faulty security mechanisms or intricate language features. Besides, more complex vulnerability types, such as *second-order vulnerabilities* or *PHP object injections*, are comparatively unknown and actively exploited by attackers.

Manual detection of complex vulnerabilities in modern PHP applications with hundreds of thousands lines of code is expensive, time-consuming, and requires deep security knowledge. With the help of static code analysis, security vulnerabilities can be detected in an automated fashion and subsequently remediated. However, previous work in this area focused only on the detection of a few traditional vulnerability types and dismissed more complex occurrences or types of vulnerabilities. Additionally, these approaches do not scale to large code bases or do not support major language features.

In this thesis, we present novel techniques designed for the efficient and precise static analysis of PHP code in order to automatically detect traditional and complex security vulnerabilities. A comprehensive configuration and simulation of over 1 200 PHP built-in features allows us to precisely model the highly dynamic PHP language. By creating block and function summaries, we are able to efficiently perform a backwards-directed taint analysis for 36 different types of vulnerabilities. More specifically, our string analysis is the first to evaluate the interaction between different types of security mechanisms, encodings, sources, sinks, markup contexts, and PHP settings. Furthermore, we are the first to detect *second-order* vulnerabilities and related *multi-step* exploits. Based on our novel forwards-directed object analysis, we are the first to automatically generate attack sequences used against *PHP object injection* vulnerabilities.

We implemented a prototype based on our approach. Our evaluation shows that it is capable of finding severe and complex vulnerabilities in modern real-world applications, previously missed by other approaches: in total, we detected 321 previously unknown vulnerabilities in 23 popular PHP applications, for example in *Joomla*, *phpBB*, and *osCommerce*. Finally, we used our prototype to study prevalent practices of developers and attackers. We first studied how developers utilize security mechanisms in practice regarding different markup contexts, and which common pitfalls exist. Then, we analyzed features and backdoors in popular PHP shells used by attackers.

Zusammenfassung

Moderne Webseiten haben sich zu interaktiven Applikationen entwickelt, die täglich vertrauliche Benutzerdaten (z. B. Kreditkartendaten und Passwörter) verarbeiten. Diese sensiblen Daten erfordern verlässlichen Schutz vor Angreifern, die Sicherheitsschwachstellen im Programmcode der Applikation ausnutzen. Vor allem Webapplikationen, die in der populärsten serverseitigen Skriptsprache PHP entwickelt wurden, sind anfällig für Schwachstellen. Trotz einer erhöhten Sensibilisierung der Entwickler für traditionelle Schwachstellentypen, wie z. B. *Cross-Site Scripting* und *SQL Injection*, treten diese weiterhin durch fehleranfällige Sicherheitsmechanismen oder missverständliche Spracheigenschaften auf. Zudem sind komplexere Schwachstellentypen, wie z. B. *Second-Order* oder *PHP Object Injection* Schwachstellen, vergleichsweise unbekannt und werden aktiv ausgenutzt.

Eine manuelle Suche nach komplexen Schwachstellen in modernen PHP-Applikationen mit mehreren hunderttausend Zeilen Code ist teuer, zeitaufwändig und erfordert Spezialwissen. Mit Hilfe von statischer Codeanalyse können Schwachstellen automatisiert erkannt werden, um sie anschließend zu beseitigen. Bisherige Ansätze in diesem Bereich konzentrieren sich jedoch nur auf die Erkennung von einigen traditionellen Schwachstellentypen und verfehlen kompliziertere Ausprägungen oder Typen von Schwachstellen. Außerdem sind die Ansätze nicht für größere Anwendungen skalierbar und wichtige Spracheigenschaften werden nicht unterstützt.

In dieser Dissertation werden neuartige Methoden für die effiziente und präzise Analyse von PHP-Code präsentiert, die es ermöglichen, sowohl traditionelle als auch komplexe Sicherheitsschwachstellen automatisiert zu erkennen. Eine umfassende Konfiguration und Simulation von über 1.200 PHP-Eigenschaften erlaubt es, die hochdynamische Sprache PHP präzise zu modellieren. Durch die Erstellung von Block- und Funktionssummarien, kann eine effiziente *Taint*-Analyse für 36 verschiedene Schwachstellentypen durchgeführt werden. Dabei wird mit Hilfe einer *String*-Analyse erstmalig das Zusammenspiel von Sicherheitsmechanismen, Kodierungen, Eingaben, Operationen, Markup-Kontexten und PHP-Einstellungen berücksichtigt. Weiterhin werden bisher unauffindbare *Second-Order* Schwachstellen und verwandte *Multi-Step Exploits* detektiert. Durch eine neuartige vorwärts gerichtete Objektanalyse können auch erstmalig mögliche Angriffsvektoren für *PHP Object Injection* Schwachstellen automatisch generiert werden.

Die neuen Analysetechniken wurden in einem Prototypen implementiert. Eine Evaluierung zeigt, dass dieser in der Lage ist, kritische und komplexe Schwachstellen in modernen Applikationen aufzuspüren, die von bisherigen Ansätzen nicht erkannt werden: insgesamt wurden 321 bisher unbekannte Schwachstellen in 23 weitverbreiteten PHP-Applikationen detektiert, u. a. in *Joomla*, *phpBB* und *osCommerce*. Abschließend wurden mit Hilfe des Prototypen gängige Vorgehensweisen von Entwicklern und Angreifern studiert. Zum einen wurde analysiert, welche Sicherheitsmechanismen von Entwicklern für welche Markup-Kontexte in der Praxis eingesetzt werden, und welche Fallstricke existieren. Zum anderen wurde untersucht, welche Funktionalitäten und Hintertüren Angreifer in populären PHP-Shells nutzen.

Acknowledgments

First and foremost, I would like to thank my supervisor Prof. Dr. Thorsten Holz for his guidance, support, and scientific insight throughout the last three years. I feel greatly privileged for receiving the opportunity to freely work on a topic of my choice in an amiable atmosphere. Likewise, my deep appreciation goes to Professor Andrei Sabelfeld for his valuable time and review. I was blessed with a great research environment at the Chair for Systems Security and with wonderful and talented colleagues. In particular, I would like to thank Felix Schuster, Jannik Pewny, Behrad Garmany, Robert Gawlik, Thomas Hupperich, and Johannes Hoffmann for an unforgettable, grandiose time!

Furthermore, I am largely indebted to Martin Bednorz, Hendrik Buchwald, Nicolas Golubovic, Nikolai Krein, and Dario Weißer for their hard work and excellent contribution to the project. For the continuous advice and assistance, I would like to extend my sincerest thanks and appreciation to Carsten Willems and Mario Heiderich. These fine gentlemen are truly inspiring and keep me motivated.

Moreover, I wholeheartedly express my gratitude to my family for their blessing, encouragement, and participation at all times. Finally, the greatest and deepest gratitude goes to Emina. Her love, understanding, help, and patience made this thesis possible.

Contents

Chapter 1

Introduction

In the following, we motivate our research on the security analysis of web applications and clarify why PHP applications are of particular interest. Then, we classify our approach in the broad research field of security analysis automation and introduce the topic of this thesis. A summary of the contributions of each individual chapter follows. Furthermore, we list our publications and provide a brief summary for each paper. At the end of this chapter, an outline for the remainder of this thesis is presented.

1.1 Motivation

Web applications evolved in the last decades from simple scripts to multi-functional applications. Today, these applications are the driving force behind the modern Web since they enable all the services with which users interact. They are used for online shopping and banking, social media and networking, email and text messaging, picture and video sharing, and even for the control of hardware and infrastructure. Often, such complex applications handle large amounts of (potentially sensitive) data, such as credit card numbers, private user data, or login credentials, that are exposed to the risks of the Internet. Cyber criminals constantly try to exploit different types of security vulnerabilities in web applications' code. Every day, thousands of websites are compromised and half a million attacks are observed [128]. A single vulnerable code line can lead to data theft, to a website infection with malicious software, to a complete takeover of the underlying web server, or to the infiltration of a company's internal network.

Thereby, the server-side programming language PHP plays a significant role. According to *W³Techs*, 81.7 % of all recognized websites use PHP, among which, popular websites are *Facebook*, *Baidu*, *Yahoo*, and *Wikipedia* [147]. Further, nine out of the ten most used content management systems are written in PHP, for example *Wordpress* and *Joomla*, which respectively run on 25 % and 3 % of all websites [146]. The downside is that security vulnerabilities are found in PHP applications above-average in comparison to other programming languages due to its weakly and dynamically typed syntax and its complex language characteristics [10]. In the MITRE CVE database, about 29 % of all security vulnerabilities found in computer software are related to PHP [88]. Vulnerabilities in widely deployed PHP software affect thousands of websites on the Web.

According to Symantec [128], the data theft announcements increased by 76 % from 2014 to 2015 and the related damage increased by 23 %, according to the Ponemon Institute [101]. As a result, the management of digital security threats moves into the global focus of the industry and politics. Companies and organisations are put to enforce new standards, policies, and laws for cyber security [35, 153]. Consequently, security researchers, web developers, administrators, and network operators around the globe seek to detect and eliminate security vulnerabilities in web – and specifically in PHP – applications. The manual detection of all possible attack vectors, however, is a challenging task since modern web applications can have several thousands lines of code and complex security vulnerabilities are hard to spot. One solution to this problem are *application security testing* tools which can detect security vulnerabilities in an automated fashion and can significantly reduce the effort, time, and costs of a manual security review [93].

1.2 Topic of this Work

In this thesis, we address several challenges for the automated security analysis of modern PHP applications. Before we introduce our analysis technique, we present various methods for the *security testing* of web applications in order to classify our work in the research field. We differentiate between *black-box* and *white-box* testing of a web application, as well as *dynamic analysis* and *static analysis* techniques [102]. Hybrid techniques are covered later in Section 3.1.

Black-box testing is a light-weight approach that is performed on the client-side of a web application without access to its source code. Here, multiple malicious input patterns for common web attacks are submitted to a deployed application in an automated fashion. At the same time, the application's responses are evaluated for abnormal behavior, such as error messages or time delays, that can indicate a security vulnerability. Although this fuzzing approach is independent of the application's programming language, it suffers from fundamental limitations. Typical drawbacks of this approach are the limited test coverage, few supported vulnerability types with a low accuracy, and the missing ability to crawl a given website "deep" enough [9, 38, 70, 84, 110].

White-box testing is performed on the server-side of an application with access to the source code. In this scenario, two distinct techniques can be applied for the automated detection of security vulnerabilities: *static application security testing* (SAST, or *static analysis*) and *dynamic application security testing* (DAST, or *dynamic analysis*). Furthermore, hybrid approaches, and intrusion detection or prevention systems exist [13, 51, 106, 124].

Dynamic analysis monitors the code of an application as it executes. In order to detect security vulnerabilities, the interpreter or the engine of the web application's programming language is modified such that it detects security violations in program execution paths [109]. While some languages are shipped with such a security feature, for example Perl and Ruby [2, 134], in the past, researchers proposed similar approaches for other languages [25, 50, 54, 77, 95]. On the one hand, dynamic analysis enables a high precision based upon actual execution and runtime data, and thus produces few false alarms. On the other hand, it can only detect software defects within the current executed program path and hence suffers from incompleteness and low performance.

Static analysis is performed solely on the source code of an application without execution. The complete source code is first transformed into an abstract model that is then analyzed for security vulnerabilities. This enables an efficient analysis with large code coverage which can be even applied to incomplete applications and different environments. As a result, SAST tools are attractive for the integration into the standard testing and the code review process in order to detect security issues as early as possible. However, static analysis is prone to false alerts due to an *abstract* analysis of data that might not conform to the runtime. More specifically, data of dynamic language features, such as *variable* variables, dynamic function calls, and dynamic code inclusions, is unknown at compile time and poses major challenges to static code analysis tools. Furthermore, while the investigation of all possible program path combinations leads to full code coverage in theory, an exponential *path explosion* can lead to performance problems [18, 78].

In the past, static security analysis of web applications has attracted a considerable amount of research [81, 112, 121, 156], specifically for the popular PHP language [61, 66, 149, 154, 156, 159]. However, recent work in this area focussed on the detection of only a limited number of *traditional* vulnerability types, such as *cross-site scripting* and *SQL injection* vulnerabilities (see Chapter 2). Moreover, the previous work is not applicable in practice for one of the following reasons:

1. *Complex* vulnerabilities that rely on multiple distinct data flows or that base on very subtle pitfalls when applying security mechanisms are missed.
2. The analysis of important language features of *modern* PHP applications, such as object-oriented code or built-in functions, is not supported.
3. The approach requires specifications or code annotations that are impractical and time-consuming.
4. The proposed approach does not scale for large PHP applications.

The goal of this thesis is to fill those research gaps and to propose a workable static analysis approach. In other words, we seek for a non-annotation based security analysis that scales to several hundred of thousands lines of code with a comprehensive coverage of language features and vulnerability types. Because the infeasibility of a perfect realization of such an approach is proven [142], the challenge is to strike the right balance between vulnerability detection, false alerts, and performance. Due to the widespread usage, we focus on the analysis of PHP applications. However, our approach can also be generalized to different languages by applying our analysis techniques to its (less diverse) features. To this end, we implemented a prototype of our novel approach and evaluated it regarding to our objective.

We complement our research on the security analysis of PHP code by performing two empirical studies based upon our prototype. We studied commonly used security mechanisms in popular PHP applications. Our analysis provides a comprehensive overview of how developers utilize security mechanisms in practice which helps us to fine-tune our static analysis engine and its results ranking. In another study, we extended our prototype implementation in order to reveal and quantify the visible and invisible features offered by malicious PHP shells. These scripts are used by adversaries on compromised machines and the detected features reveal common practices of attackers after vulnerability exploited.

1.3 Contributions

We believe that our work is of great value for designers of static code analysis tools, especially since the basic insights can be also applied to programs implemented in other languages. Additionally, developers and security researchers can deepen their security knowledge about the PHP language and its pitfalls, as well as about common vulnerability types and exploitation. In the following, we introduce the contributions of each chapter of our thesis.

Chapter 2: Background on Vulnerabilities in PHP Applications

In Chapter 2, we provide a comprehensive overview of the security landscape of PHP code. We demonstrate that a precise understanding of the complex characteristics of the PHP language is essential to detect security vulnerabilities in modern PHP applications. For this purpose, we introduce several dynamic languages features and their intricacies. Furthermore, we survey common mechanisms to sanitize and validate data in PHP applications and highlight the pitfalls that can occur. Next to traditional vulnerability types, we study the problem of second-order vulnerabilities and multi-step exploits. We survey related *persistent data stores* that can be used for the intermediate storage of an attack payload before it is used in a sensitive operation. Finally, we perform a systematic analysis of *PHP object injection* (POI) vulnerabilities and demonstrate how such vulnerabilities can be exploited via *Property-Oriented Programming* (POP), a variant of code reuse attacks against web applications.

Chapter 3: Designing a Static Code Analysis Tool

In the third chapter, we introduce the algorithms of our tool which is dedicated to the specifics of the PHP language and is the first to perform a fine-grained analysis of a large number of PHP built-in features. We are the first to support the detection of 36 different types of security vulnerabilities. Our novel approach of combining demand-driven backwards-directed taint analysis with string analysis enables a refined context-sensitive vulnerability confirmation with respect to the interaction of sink, source type, sanitization, validation, encoding, and PHP configuration. To this end, we are able to precisely analyze input sanitization and validation mechanisms, a crucial step to lower the number of potential false positives and negatives. Furthermore, we are the first to propose an automated approach designed to statically analyze object-oriented PHP code. More specifically, we introduce our lightweight object- and field-sensitive data flow analysis that scales to large PHP applications. As a result, we are the first to present an automated approach to statically detect POI vulnerabilities in object-oriented PHP code and to automatically verify the severity by constructing exploitable gadget chains. Additionally, we are the first to propose an automated approach to statically analyze second-order data flows through databases, file names, and session variables by using string analysis. This enables us to detect second-order and multi-step exploitation vulnerabilities in web applications.

Chapter 4: Evaluation of a Prototype Implementation

We implemented a prototype of our approach. In the fourth chapter, we evaluate our approach on large, real-world applications and demonstrate that our prototype is capable of finding several previously known and unknown, severe vulnerabilities. We measure the performance results and present the efficiency of our approach. Furthermore, we compare our results to previous work in this area and demonstrate that our prototype outperforms state-of-the-art tools. Besides traditional taint-style vulnerabilities, we evaluate second-order data flows of six real-world web applications. As a result, we detect 159 previously unknown second-order vulnerabilities ranging from XSS to *remote code execution* attacks. Furthermore, we evaluate our approach for the detection of *PHP object injection* vulnerabilities for 10 well-known and recently affected applications. To this end, we detect 30 new POI vulnerabilities and 28 new gadget chains. In total, our prototype detected 321 previously unknown vulnerabilities with an average false discovery rate of 20 %.

Chapter 5: Empirical Studies

In Chapter 5, we present our approach to enumerate security mechanisms in PHP applications and features in PHP shells via static code analysis techniques. We evaluate our approach to number security mechanisms with 25 popular real-world applications. To the best of our knowledge, this is the largest study on the usage of security mechanisms in modern PHP applications. Our study proposes answers to the research questions a) which security mechanisms are used how often in modern (web) applications, b) which security mechanism is used to prevent which vulnerability type in which markup context, and c) which pitfalls occur in practice. Additionally, we evaluate our approach to enumerate features in 481 popular malicious PHP shells. Our manual analysis reveals how many PHP shells contain a backdoor. This is the first comprehensive study of web shells and our study provides novel insights about the visible and invisible features of attack tools used after a successful web server compromise.

1.4 List of Publications

The work on our static analysis prototype presented in this thesis resulted in various academic peer-reviewed publications. In the following, we list our publications chronologically and provide a brief summary of their topics.

Simulation of Built-in PHP Features for Precise Static Code Analysis

In our initial paper, we presented the basics of our static analysis approach. We showed that a fine-grained analysis of built-in PHP features is the key for detecting complex vulnerabilities in modern PHP applications. As a solution, we proposed our approach on precisely modeling the highly dynamic PHP language. An evaluation of a prototype implementation showed that we outperform state-of-the-art tools. The paper was published with Thorsten Holz at the *NDSS'14* conference.

Static Detection of Second-Order Vulnerabilities in Web Applications

Next, we extended our prototype with the capability of detecting second-order vulnerabilities. These vulnerabilities occur when an application stores an attack payload on the web server and then later on uses it again in a security-critical operation. Our presented technique is the first automated static code analysis approach and is also applicable to different programming languages. The work was published together with Thorsten Holz at the *23rd USENIX Security Symposium* and was awarded with the *Internet Defense Prize* by *Facebook*.

Code Reuse Attacks in PHP: Automated POP Chain Generation

In this follow-up paper, we are the first to propose analysis methods for the automated generation of gadget chains used in *PHP object injection* vulnerability attacks. Similar to code reuse attacks for memory corruption vulnerabilities, reusing existing code fragments is a viable attack vector when an attacker is able to inject arbitrary objects into a PHP application. The paper was published in a joint work with Nikolai Krein and Thorsten Holz at the *CCS'14* conference and received the *best student paper* award.

Experience Report: An Empirical Study of PHP Security Mechanism Usage

Based on our experience on the static analysis of PHP application vulnerabilities, we studied commonly used input sanitization and validation mechanisms and their pitfalls in 25 popular PHP applications. We used our prototype to analyze 2.5 millions lines of code and found certain markup contexts and security mechanisms being more frequently vulnerable than others. This joint work with Thorsten Holz was published at the *ISSTA'15* conference.

Security Analysis of PHP Bytecode Protection Mechanisms

In this paper, we proposed a method to automatically recover the original source code of protected PHP applications. We analyzed the inner working of commercial products, such as *ionCube*, *Zend Guard*, and *SourceGuardian*, and we introduced a generic approach on the decompilation of obfuscated PHP bytecode. We then used our static analysis prototype to detect backdoors and vulnerabilities in the recovered source code. The work was published together with Dario Weißer and Thorsten Holz at the *RAID'15* conference.

No Honor Among Thieves: A Large-Scale Analysis of Malicious Web Shells

In this work, we performed the first comprehensive study of PHP shells—backdoors uploaded by attackers on compromised web servers in order to maintain their access. With the help of our static analysis prototype, we discovered and quantified the visible and invisible features provided by popular shells. A manual analysis of authentication mechanisms revealed that about one third of those can be bypassed. Furthermore, we used dynamic analysis and honeypots to analyze outgoing *homephoning* and incoming attacker traffic. The joint work with Oleksii Starov, Syed Sharique Ahmad, Thorsten Holz and Nick Nikiforakis was published at the WWW'16 conference.

1.5 Outline

The remainder of this thesis is structured as follows. In Chapter 2, we first provide a systematic background on vulnerabilities in PHP applications. We present features and intricacies of the PHP language which can lead to different types of vulnerabilities and pitfalls when applying security mechanisms. With this thorough background knowledge on PHP application vulnerabilities, we begin to design a static code analysis tool for the automated vulnerability detection in Chapter 3. We introduce our design approach and show how we handle the challenges of analyzing modern PHP applications efficiently, as well as how we are able to detect even complex vulnerability types. An evaluation of a prototype implementation of our approach is then presented in Chapter 4. In this chapter, we present characteristic vulnerability findings, discuss false positives, and compare our results to related work in the field. After we looked at security vulnerabilities and their automated detection, we used our prototype implementation to empirically study their nature. The results are presented in Chapter 5. First, we study and investigate different protection mechanisms that are applied by developers in order to prevent security vulnerabilities. Then, we study features in PHP shells that are commonly used by attackers after successfully exploiting a security vulnerability. Finally, we conclude this thesis with Chapter 6, where we summarize the presented material and outline topics for future research.

Chapter 2

Background on Vulnerabilities in PHP Applications

The server-side scripting language PHP was incarnated by Rasmus Lerdorf in 1994. It started as a simple suite of CGI binaries, referred to as *personal home page* (PHP) tools, written in the C programming language. Since the public release of its source code in 1995, the tools were expanded and rewritten, turning into a new, independent programming language for web developers with the recursive acronym *PHP: Hypertext Preprocessor*. Due to an easy adoption and a large number of built-in features PHP gained popularity: the PHP interpreter was installed on 10 % of all web servers in 1998, and runs on over 80 % of all web servers today [147]. However, the early PHP was not designed to be a programming language. Instead, it grew organically and went through multiple redevelopments [131]. In practice, language inconsistencies, insecure default settings, missing type information, and a missing specification led to many bugs and vulnerabilities in PHP applications.

In this chapter, we first introduce a variety of language features that are characteristic for the PHP language (Section 2.1). It provides an overview of the challenges one has to overcome, when designing a tool for the automated analysis of PHP source code. We then introduce taint-style vulnerabilities in Section 2.2, the most prevalent security vulnerabilities in PHP applications. By looking at codes vulnerable to different types of vulnerabilities, we can develop a generic pattern for the automated detection. In Section 2.3, we elaborate security mechanisms that are applied by developers in order to prevent the introduced vulnerabilities. The correct identification and analysis of these mechanisms is indispensable to distinguish between an exploitable security flaw and safe code. Furthermore, we introduce related programming mistakes due to common pitfalls that can base on the intricacies of the language and still lead to vulnerabilities. At the end of this chapter, we introduce two advanced vulnerability types. In Section 2.4, we introduce second-order vulnerabilities. These occur when an attack payload is first stored by the application on the web server and then is later on used in a security-critical operation. In Section 2.5, we introduce *PHP object injection* vulnerabilies. We show that code reuse attacks are a viable attack vector against web applications, and how they can be exploited via *property-oriented programming*.

2.1 Intricacies of the PHP language

PHP is a highly dynamic language with lots of complicated semantics [12]. While these are very attractive for quick and easy application development, they hide serious security risks and pitfalls. In this section, we introduce complex language features commonly used in PHP applications. Understanding these features plays a key role for the precise detection of security vulnerabilities in PHP code and for the development of a static analysis tool that can correctly handle these language features.

Dynamic and Weak Typing

PHP is a *dynamically typed* language [136]. For variables this means they do not have to be *declared* explicitly. The variable type depends on the first defined value assigned to it at runtime. Additionally, PHP is a *weakly typed* language and its variables are not *bound* to a specific data type. Thus, data types can be mixed with other data types at runtime. For example, in Listing 2.1, the string "test" in line 2 is evaluated to the number `0` in order to fit the addition with the number `1` in line 3. The result of type integer is stored in the variable `$var2` whose previous data type was string.

```
1 $var1 = 1;
2 $var2 = 'test';
3 $var2 = $var1 + $var2; // 1
```

Listing 2.1: Addition of a string and an integer.

Variable Variables

In PHP, variables are usually introduced with the dollar character followed by an alphanumerical, case-sensitive name. However, a variable name can also be an expression, for example, the value of another variable or the return value of a function call that is only known at runtime. This makes it extremely difficult to analyze the PHP language statically.

```
1 $name = "x";
2 $x = "test";
3 echo $$name; // test
4 $y = ${getVar()};
```

Listing 2.2: Variable variables in PHP.

In Listing 2.2, the variable `$x` is accessed dynamically in line 3 and the string *test* is printed. The variable assigned to the variable `$y` in line 5 is also generated dynamically at runtime and uses the return value of the user-defined function `getVar()`.

Dynamic Arrays

PHP arrays are hash-tables which map numbers or strings (referred to as *keys*) to values. The values of a user-defined array are initialized with the `array` operator, as demonstrated in line 2 of Listing 2.3. If the name of the key is omitted, the value is associated with a numerical key which is incremented by `1`, starting from `0` or the highest occurring numerical key.

```
1 $key = 6;
2 $arr = array('a', "4" => $key, 'foo' => 'c', 'd');
3 $arr[] = 'test';
4 // Array ( [0] => a [4] => 6 [foo] => c [5] => d [6] => test )
5 print $arr[$key]; // test
```

Listing 2.3: Dynamically generated key names in an array.

The fourth line in Listing 2.3 shows the generated array structure. When analyzing this code statically it is unclear what value corresponds to the key *6* because its name is not explicitly defined in the code and only generated at runtime.

Dynamic Constants

Moreover, it is possible to define *constant* scalar values as in other programming languages like *C*. However, the constant name can be dynamically defined by PHP's built-in function `define()` and dynamically accessed by the built-in function `constant()`. The dynamic definition and access of constants is demonstrated in Listing 2.4.

```
1 define("FOO", $_GET['a']);
2 $a = constant('F' . 'OO');
```

Listing 2.4: Dynamic constants in PHP.

Although a constant may not change once it is defined, it is possible to conditionally define constants throughout the program flow or to generate these dynamically with user input. Additionally, PHP is shipped with predefined (called *magic*) constants of the PHP core which mainly represent the current PHP configuration.

Dynamic Functions

Also, user-defined functions are not completely static language constructs. Several functions with the same name can be defined conditionally by the developer. Thus, a totally different function may be called, depending on the program flow. It is also possible to define a function `B()` within another function `A()` that is only present during the execution of `A()`. Furthermore, PHP's built-in functions `func_get_arg()` and `func_get_args()` allow to dynamically fetch parameters that are passed to the function.

```
1 $name = 'step' . (int)$_GET['id'];
2 $name();
3 array_walk($arr = array(1), $name);
```

Listing 2.5: Dynamically build and executed function name.

Listing 2.5 illustrates two different possibilities to *call* a function dynamically. The function name is built dynamically in the first line where the user supplies a numerical `id` which is appended to the string *step*. The created function name is only known at runtime. In line 2, the function is called by adding parenthesis to the variable `$name`.

In the third line, the dynamically build function name is used as a *callback* function. The PHP built-in function `array_walk()` executes the *callback* function in the second argument for every element in the array passed to its first argument. Besides, the built-in function `create_function()` allows to create function code dynamically.

Dynamic Code

The `eval` operator and the less known function `assert()` allows to directly evaluate PHP code that is passed as string to its first argument. Dynamically generated code is very challenging to analyze if the executed PHP code is only known at runtime and cannot be reconstructed statically.

```
1 preg_replace("/\\x[a-f0-9]{2}/ie", "chr(hexdec(\\1))", $str);
```

Listing 2.6: Dynamic code in PHP.

Another example for dynamic code evaluation is shown in Listing 2.6. Here, the function `preg_replace()` performs a regular expression search defined in the first argument and replaces matched strings with the string given in the second argument. The regular expression can be specified with several modifiers after the expression delimiter (`/`). When the modifier *e* is used, the replacement string is evaluated as PHP code before the matches are replaced. This behavior can lead to serious security vulnerabilities when not used with caution [83].

Dynamic Includes

The code of large PHP projects is often split into several files and directories. At runtime, the code can be connected and executed conditionally. The PHP operator `include()` opens a specified file, evaluates its PHP code, and returns to the last instruction after the `include()` operator. This operator can also have a return value if the included code has a `return` statement. Furthermore, the file name of the inclusion operators can be build dynamically which is very challenging to statically reconstruct in complex web applications. During static code analysis, it is crucial to handle all file inclusions in order to analyze the PHP code correctly. A single file inclusion can import vulnerable parts of the application or initialize data that can change the program flow significantly. Additionally, a security vulnerability can occur when user input is used within the file path (see Section 2.2.3).

Built-in Functions and Sinks

Depending on the configuration and version, PHP is shipped with several *extensions* that provide built-in functions to the developer [45]. By default, PHP 5.4.8 is compiled with 24 *core extensions* that provide 732 basic built-in functions, for example for string and array processing, mail and file handling, or parsing date and time formats. Additionally, there are 81 *bundled extensions* which are shipped with PHP although not enabled by default. These include 2 005 built-in functions mainly to connect to FTP, LDAP, and DBMS servers, or parse additional file formats, such as XML and ZIP. Furthermore, PHP can be extended with *external extensions* that require external libraries. In total, 228 extensions with 5 701 built-in functions are documented [46]. Static code analysis is losing precision whenever a built-in function is called that is not configured in the tool. More precisely, built-in functions that perform security critical operations on the web server must be configured as *sensitive sinks*.

Superglobals and Sources

Superglobals are built-in arrays, initialized from the PHP interpreter and available in all *scopes*. They allow quick access to the HTTP request header, environment, and global scope. The most commonly processed HTTP header fields are the GET and POST parameters. These are the *visible* parameters entered by the user in the browser's address bar or in an HTML form field. In PHP, the received user input is stored in the `$_GET` and `$_POST` array. Cookies can be altered by a user in the browser configuration or by crafting an HTTP request manually. In PHP, the cookie values are stored in the `$_COOKIE` array. The superglobal `$_REQUEST` holds a copy of all the values in `$_GET`, `$_POST`, and `$_COOKIE`. When a file is uploaded, the file name is stored in the superglobal `$_FILES` array. These superglobals are potential *sources* of user input.

Furthermore, a user can modify all the other HTTP header names and values by crafting an HTTP request manually. In PHP, all header fields that appear as "name: value" pair separated by a colon are stored in the `$_SERVER` array with a `HTTP_` prefix followed by the header name (dashes are replaced with underscores). Developers are often not aware that these names and values can be altered by an attacker next to the usual *visible* parameters of the application. Specifically, the predefined `$_SERVER` keys `PHP_SELF` and `REQUEST_URI`, which contain the requested file path, and the header field `HTTP_HOST`, which contains the host's name, are assumed as static values but can be altered by an attacker. A static code analysis tool needs to configure all possible *sources* of malicious user input in order to detect all possible entry points for attackers.

Register Globals

Before PHP version 4.2.0, the PHP setting `register_globals` was activated by default. It provides an attacker with even more entry points and introduced many security issues [47]. The setting allows an attacker to initialize any global variable via an HTTP request parameter. This can lead to an unexpected behavior and security issues, for example, when the GET request to `index.php?loggedin=1` initializes the variable `$loggedin` in *index.php* to the value *1* and this variable is then used within an authentication check. Thus, the `register_globals` directive was deactivated by default in later PHP versions and was removed as of PHP 5.4.0.

Still, the same behavior can be achieved by using PHP built-in features, as shown in Listing 2.7. For example, the built-in functions `extract()` (line 1) and `parse_str()` (line 2) can be used to populate values in an array or URL string into the global scope. When these functions are called with user input and no additional arguments, the `register_globals` directive is, effectively, simulated. The same applies to a call to the built-in function `import_request_variables()` (line 3). Moreover, each key/value pair within the superglobal `$_GET` or `$_POST` array can be populated into global variables by constructing a loop and assigning each key to a variable dynamically (line 4).

```
1 extract($_REQUEST["c99shcook"]);
2 parse_str($_SERVER['HTTP_REFERER']);
3 import_request_variables("GPC");
4 foreach($_GET as $k => $v) { $$k = $v;}
```

Listing 2.7: Variants for simulating `register_globals`.

2.2 Taint-style Vulnerabilities

A security vulnerability occurs when data supplied by the user is utilized in a security critical operation of a web application and was not sufficiently sanitized beforehand. An attacker might be able to *exploit* this issue by injecting malicious input that can change the behavior or the result of this operation [103]. This kind of vulnerability is called *taint-style vulnerability* because an untrusted *source*, such as user-supplied data via an HTTP parameter or header, is considered as *tainted* and literally flows into the vulnerable part of the program (referred to as *sensitive sink*) [24, 81, 94, 109]. In this section, we survey three common *taint-style* vulnerabilities. By examining the similarities of these vulnerabilities, a general pattern for our static analysis can be created.

2.2.1 Cross-Site Scripting

Cross-Site Scripting (XSS) is the most common security vulnerability in web applications [73]. It occurs when user input is reflected unsanitized to the HTML result of the application. An attacker can then inject HTML markup into the response page rendered by the client's browser. Commonly, this attack is performed by crafting a malicious link that is sent to a victim. When opened, the attacker's JavaScript payload within the link is reflected by the application and executed in the victim's browser in the context of the web application's domain. This enables the attacker to perform phishing attacks or to steal cookies associated with the domain.

```
print('<h1>Welcome ' . $_GET['name'] . '</h1>');
```

Listing 2.8: Unsanitized output of user input.

Listing 2.8 shows a simple PHP application that is vulnerable to XSS. A user can specify a `name` in the GET parameter list that is printed to the HTML result page by the application. Instead of choosing an alphanumerical name, a malicious user can inject meta characters. Because these characters are embedded unfiltered to the HTML page by the application, they are interpreted by the browser as HTML markup. Listing 2.9 shows the HTML result page in case a user injects HTML via the GET parameter *name*.

```
/index.php?name=foo</h1><script>alert(1)</script>

  <h1>Welcome foo</h1><script>alert(1)</script></h1>
```

Listing 2.9: The HTML response for injected markup.

The browser renders the HTML code and displays the headline *Welcome foo*. Then it renders the injected `script` tags and the JavaScript is executed by the browser's Javascript engine. Beside a harmless alert box, the attacker could access the user's session cookie and forward it to an attacker controlled domain. Then, the attacker can use the stolen cookie to hijack the user's session and to control his user account. In order to patch this vulnerability, the output can be sanitized (see Section 2.3.2). Meta characters, such as `<` and `>`, as well as quotes must be replaced by their corresponding HTML entities `<` and `>`. Then, the characters are still displayed by the browser but not rendered as HTML.

2.2.2 SQL Injection

Web applications are often connected to a database that handles large data sets. Sensitive data, such as credit card numbers, login credentials, or private messages, is usually stored in databases. If a web application dynamically generates a SQL query with unsanitized user input, an attacker can inject arbitrary SQL syntax and modify the SQL query. This vulnerability is called *SQL injection* (SQLi) [52]. Depending on the environment, the attacker is able to extract sensitive data from the database, modify data, or compromise the web server, for example, by abusing file functionalities of the SQL syntax.

```
1 $name = $_GET['name'];
2 $query = "SELECT phone FROM users WHERE name = '$name'";
3 $result = mysql_query($query);
```

Listing 2.10: SQL query build with unsanitized user input.

Listing 2.10 shows the source code of a PHP application that is vulnerable to SQLi. A user can specify a name by the GET parameter `name` (line 1) that is embedded unsanitized into a SQL query (line 2). The query requests the column `phone` from the table `users` corresponding to the supplied `name`.

An attacker is able to extract the users' passwords by modifying the SQL query. The SQL operator `UNION` allows to stack several `SELECT` statements within one SQL query. It allows to append another `SELECT` statement that requests data from the `password` column of the table `users`. The injected payload and the resulted SQL query is shown in Listing 2.11.

```
/index.php?name=foo'+UNION+SELECT+password+FROM+users-- -

  SELECT phone FROM users WHERE name = 'foo' UNION SELECT password FROM users-- -'
```

Listing 2.11: SQL query with injected SQL code

The attacker breaks out of the `name` string with a single quote followed by her injected SQL syntax. The double dashes at the end of the query introduce a SQL comment so that the trailing single quote is ignored. Instead of a phone number, the appended `SELECT` query returns all passwords from all `users`. In order to prevent SQL injection prepared statements can be used (see Section 2.3.2). The vulnerability can also be patched by sanitizing the user input before it is embedded into the query. In other words, all quotes must be escaped within a quoted string so that breaking out of the quotes and taking control over the SQL query is infeasible for an attacker.

2.2.3 File Inclusion

As introduced in Section 2.1, the path name for inclusion of a PHP file can be dynamically constructed. A *file inclusion* vulnerability occurs when user input is embedded unsanitized into the file path used for inclusion. It allows an attacker to tamper the file path and to force the application to include unexpected files into the execution process. Listing 2.12 shows two different file inclusion vulnerabilities. Similar to XSS and SQLi, the exploitation depends on the markup around the injection point which in this case is represented by the constant parts of the file path.

```
1 include($_GET['file']);                  // ?file=http://attacker.com/shell.txt
2 include('includes/' . $_GET['inc']);     // ?inc=../../../../../../etc/passwd
```

Listing 2.12: Dynamic file inclusions with user input.

In the first line, none of the file path parts are hardcoded in the application and the vulnerability is known as a *remote file inclusion* (RFI) vulnerability: an attacker can inject different protocol handlers, such as `http://` or `ftp://`, in order to include remote files. Furthermore, it is possible to include local files and to disclose the source code of the PHP application by using the `php://` protocol handler. In order to use protocol handlers within a file inclusion, the PHP setting `allow_url_include` has to be enabled. It is disabled by default since PHP 5.2.0.

In the second line of Listing 2.12, the directory `includes` is prefixed to the file path and the vulnerability is known as a *local file inclusion* (LFI) vulnerability. The path prefix prevents to inject a protocol handler at the beginning of a path. However, an attacker can still use a *path traversal* attack in order to include local files from the file system. For this purpose, the attacker can use the characters `../` in order to traverse out of the *includes* directory and to access sensitive files located above in the file system. In case the attacker is able to write PHP code to the file system, for example through a log file, this can lead to *remote code execution*. In order to prevent a *path traversal* attack, the built-in function `basename()` can be used to eliminate injected path names and to limit the user input to file names.

2.2.4 Other Vulnerability Types

There are many other types of taint-style vulnerabilities [98]. The high amount of built-in functions in PHP suggest that these vulnerabilities can occur in a wide diversity. For example, the vulnerabilities *LDAP injection*, *XPath injection*, and *XQuery injection* are very similar to SQLi, where the attacker can inject into a query of another resource. An *arbitrary file delete*, a *directory listing*, or a *file disclosure* vulnerability is very similar to LFI, where an attacker injects into a file path used for a different file operation. Other vulnerability types, such as a *session fixation*, *reflection injection*, or an *open redirect*, have a more unique attack pattern. However, all taint-style vulnerabilities rely on the same principle: an unsanitized source of data type *string* flows into a sensitive sink and allows malicious users to change the behavior and actions of the sensitive sink.

2.3 Security Mechanisms

As introduced in the last section, user input that is processed in a security sensitive operation of the web application can cause a vulnerability, such as XSS, SQLi, or RFI. In order to prevent such critical vulnerabilities, the user input has to be sanitized or validated beforehand. For this purpose, a *security mechanism* is applied between the user input (*source*) and the sensitive operation (*sink*) so that malicious data cannot reach the sensitive operation.

In this section, we study the variety of security mechanisms used by developers in practice. Although the possible ways of implementation are endless, our list covers all

different mechanisms we experienced in real-world applications. The mechanisms can be grouped into *input sanitization* (see Section 2.3.1) and *input validation* (see Section 2.3.3) mechanisms. Some sanitization and validation mechanisms have to be applied carefully to the *context* of the markup (see Section 2.3.2 and Section 2.3.4). Additionally, a security mechanism can be applied *path sensitively* (see Section 2.3.5). In the following, these mechanism are illustrated with an example and common pitfalls are highlighted.

2.3.1 Generic Input Sanitization

Generally speaking, during input *sanitization* the data is transformed so that harmful characters are removed or defused. The advantage of this approach is that relevant parts of the data stay intact, while only certain markup characters are removed or replaced. This way, the application can proceed with the sanitized data without a request for resubmission. In the following, we present several methods to sanitize data *generically* against all type of injection flaws, such as XSS and SQLi.

Explicit Typecasting Numeric characters can be safely used in security sensitive string operations. In order to ensure that only numeric characters are present, a string can be typecasted to a number by using the *typecast* operator or related built-in functions.

```
1 $var = (int)$var;       // safe
2 $var = intval($var);    // safe
3 settype($var, 'int');   // safe
```

Listing 2.13: Examples for explicit typecasting.

All three operations in Listing 2.13 ensure a secure use of the variable `$var` regarding injection flaws. PHP uses *duck typing* to determine the integer value of a string. An empty and non-empty string is typecasted to the number 0. However, if the string starts with a number (*"123abc"*), the number is used as result of the typecast (*123*). We will later on introduce pitfalls that are associated with duck typing.

Implicit Typecasting Similar to an explicit typecast, an implicit typecast occurs automatically when data is used in mathematical operations. Listing 2.14 shows in line 1 an addition in which `$var` is safely typecasted to *int* before a number is added.

```
1 $var = $var + 1;   // safe
2 $var = $var++;     // unsafe
```

Listing 2.14: Examples for implicit typecasting.

On the contrary, the *increment* operator in line 2 does not perform a typecast and it also works on strings. For example, the last character in the string *aaa* will be incremented to *aab*. Thus, `$var` in line 2 can still contain malicious characters.

Formatting Typecasting is also performed by PHP's built-in *format string* functions. Different *specifiers* (beginning with a percentage sign) can be used in a format string where they determine the type of the data with which they will be replaced. An example that uses the specifier `%s` (string) and `%d` (numeric) is given in Listing 2.15.

```
1 $var = sprintf("%s %d", $var1, $var2);   // unsafe / safe
```

Listing 2.15: Sanitization with a format string function.

The argument `$var1` is unsafely embedded into the string assigned to `$var`. Contrarily, `$var2` is safely typecasted to *int* before it is embedded into the string.

Encoding The exploitation of injection flaws usually requires special characters. Thus, next to numbers, alphabetical letters can be considered to be a safe character set. By encoding data to an alphanumerical character set, the data is sanitized. Listing 2.16 provides a few encoding examples.

Although the *base64* and *url* encoding (line 1 and 2) introduce a few special characters (`+`, `/`, `=`, or `%`), these are generally insufficient to form a malicious payload. Hence, these encodings can be considered as safe when used in a sensitive sink. Other encodings, however, include the full set of ASCII characters in the transformed output and are thus unsafe to use in sinks (line 3). Specifically, the transformation or *decoding* to the original data is unsafe because it reanimates malicious characters (line 4).

```
1 $var = base64_encode($var);      // safe
2 $var = urlencode($var);          // safe
3 $var = zlib_encode($var, 15);    // unsafe
4 $var = urldecode($var);          // unsafe
```

Listing 2.16: Transforming data into different encodings.

Filtering It is also possible to sanitize data by built-in filter functions. If the data passes a filter, it is then returned unmodified. Otherwise, *false* is returned so that the function can also be used for input validation (see Section 2.3.3). Listing 2.17 demonstrates the usage of two filter functions.

```
1 $var = filter_var($var, FILTER_VALIDATE_INT);    // safe
2 $var = filter_var($var, FILTER_VALIDATE_EMAIL);  // unsafe
3 $vars = array_filter($vars, 'is_numeric');       // safe
4 $vars = array_filter($vars, 'is_file');          // unsafe
```

Listing 2.17: Sanitization with a filter.

While the filter for integer/numeric values is safe (line 1 and 3), filtering for valid email addresses or files (line 2 and 4) is not because the character set of an email or file name allows special characters. For example, the SQL injection payload `1'or'1'--@abc.com` represents a valid email and file name.

2.3.2 Context-Sensitive Input Sanitization

In contrast to generic input sanitization, context-sensitive sanitization removes or transforms only a small set of special characters in order to prevent exploitation of a specific vulnerability type or a subset of vulnerabilities. Therefore, sanitized data may still cause a vulnerability when used in the wrong markup context or another type of sensitive sink. Again, in the following, we provide examples for security mechanisms and common pitfalls inspired by real-world code we found.

Converting A common method to distinguish between HTML markup characters and data is to convert markup characters within data to HTML entities [63]. In Listing 2.18, the built-in function `htmlentities()` is applied to different HTML contexts.

```
1 $var = htmlentities($var);
2 echo '<a href="abc.php">'.$var.'</a>';           // safe
3 echo '<a href="abc.php?var='.$var.'">link</a>';  // safe
4 echo "<a href='abc.php?var=".$var."'>link</a>"; // unsafe
5 echo "<a href=abc.php?var=".$var.">link</a>";   // unsafe
6 echo '<a href="'.$var.'">link</a>';             // unsafe
```

Listing 2.18: Converting meta characters to HTML entities.

The function `htmlentities()` converts the < and the > characters to the entities `<` and `>`, as well as the double-quote character `"` to `&quot;`. Thus, the data is safely used in line 2, where no new HTML tag can be opened with a < character, and in line 3, where no double-quote can be used in order to break the *href* attribute. Nevertheless, when single-quotes (line 4) or no quotes (line 5) are used for the attribute, an attacker can inject eventhandlers in order to execute JavaScript code. In line 6, double-quotes are used and cannot be broken, however, a JavaScript protocol handler can be injected at the beginning of the URL attribute that crafts a malicious link.

Escaping In SQL markup, string values are *escaped* in order to prevent breaking the quotes in which the value is embedded. A prefixed backslash before a quote tells the SQL parser to interpret the next quote as data instead of syntax.

```
1 $var = addslashes($var);
2 $sql = "SELECT * FROM user WHERE nr = '".$var."'";  // safe
3 $sql = 'SELECT * FROM user WHERE nr = "'.$var.'"';  // safe
4 $sql = "SELECT * FROM user WHERE nr = ".$var;       // unsafe
5 mysql_query($sql);
```

Listing 2.19: Escaping data for a SQL query.

In Listing 2.19, line 1, a variable is escaped with the built-in function `addslashes()`. This function prevents the breaking of a single- or double-quoted string value (line 2 and line 3). However, when no quotes are used in the SQL query (line 4), breaking quotes is irrelevant since an attacker can append SQL syntax directly. Thus, the query is vulnerable to SQLi although a security mechanism is in place.

Furthermore, truncating a string *after* it was escaped introduces a security risk. If the truncation happens at the position of an escaped character (`\'`), a backslash remains unescaped at the end of the string (\) that breaks any upcoming quote within the query.

Preparing A safer way to separate data and SQL syntax is to use *prepared statements* [135]. In Listing 2.20, the SQL statement is prepared with place holders for parameters. Data can then be bound to each place holder which will be safely inserted at runtime, regardless of the type of quote or data. Still, if the SQL statement is prepared dynamically, it can be vulnerable to SQL injection. For example, in line 1, SQL syntax can be injected through the variable `$pfx` into the query. Another pitfall to consider is that the *name* inserted to the table *user* can still cause a *second-order* vulnerability [31].

```
1 $stmt = $db->prepare("INSERT INTO ".$pfx."user (id, name) VALUES (?, ?)");
2 $stmt->bind_param("i", $var); // safe
3 $stmt->bind_param("s", $var); // safe
4 $stmt->execute();
```

Listing 2.20: Binding parameters to a prepared statement.

Replacing The manual replacement of malicious characters is error-prone in practice. Listing 2.21 shows two ways of replacing single-quotes that look safe at first sight.

```
1 $var = str_replace("'", "", $var);    // unsafe
2 $var = str_replace("'", "\'", $var);  // unsafe
3 $sql = "INSERT INTO user VALUES ('".$var."','".$var."')";
4 mysql_query($sql);
```

Listing 2.21: Two examples for manual escaping.

In line 1, single quotes are removed completely while in line 2 they are escaped with a backslash. However, the backslash itself is forgotten in both replacements. Hence, a backslash can be injected to break the single quotes. The second replacement will replace the code `\'` with `\\'`, which escapes the backslash and leaves the single quote unescaped.

Regex Replacing Regular expressions (*regex*) can be used for string replacement and are error-prone if not specified carefully [8]. For example in Listing 2.22, all characters except for those specified in brackets shall be removed to ensure safe data output.

```
1 $var = preg_replace("/[^a-z0-9]/", "", $var); // safe
2 $var = preg_replace("/[^a-z.-_]/", "", $var); // unsafe
3 echo $var;
```

Listing 2.22: String replacement with regular expressions.

The first regular expression allows alphanumerical characters and is safe. The second regular expression could intent to allow lowercase letters as well as the *dot*, *minus*, and *underscore* character. However, the full ASCII range between the *dot* and *underscore* character is allowed, including the characters < and > that allow to inject HTML.

2.3.3 Generic Input Validation

Next to input *sanitization* that transforms data into a safe character set, data can be simply refused if it does not hold a condition or if it fails a check. This input *validation* ensures that only data which already consists of a safe character set can reach a sensitive sink while data containing malicious characters is refused. In the following, we introduce generic conditions and checks in order to validate data against all types of injection flaws that we empirically found during our analysis.

Null Validation The easiest way to validate that no malicious character is within a given string is to check if it is empty or not set (see Listing 2.23). However, this also implies that no data can be used. A null validation is commonly used in combination with a previously executed `unset()` operation. A static code analysis tool should be able to calculate the boolean logic behind a `not` operator and multiple `else` or `elseif` branches (line 4).

```
1 if(empty($var)) { }           // safe
2 if(!isset($var)) { }          // safe
3 if(!$var) { }                 // safe
4 if(empty($var)) { } else { }  // unsafe
```

Listing 2.23: Validating a variable's initialization.

Type Validation Validation can also be performed by checking the data type. Listing 2.24 shows four examples that check for a numerical data type. In line 3, PHP's *duck typing* is used when a string is provided for an integer typecast. According to its rules, the typecast result of a string that starts with a number will bypass the validation. The same applies to line 4, although `$var` is sanitized because it is overwritten with the typecast result.

```
1 if(is_numeric($var)) { }        // safe
2 if(is_int($var) === true) { }   // safe
3 if((int)$var) { }               // unsafe
4 if($var = (int)$var) { }        // safe
```

Listing 2.24: Validating a variable's type.

Format Validation Next to the data type, a specific data format can be enforced. For example, the time and date format ensures that no malicious payload can be crafted with the given set of characters (see Listing 2.25). Other formats, however, can allow malicious characters, such as parts of the URL format.

```
1 if(checkdate($var)) { }           // safe
2 if($var = strtotime($var)) { }    // safe
3 if($vars = parse_url($var)) { }   // unsafe
```

Listing 2.25: Validating a variable's format.

Comparing By comparing input against a specific non-malicious value, the data is implicitly limited to this value. In PHP, this can be done by the *equal* operator, the *identical* operator, or by built-in functions (see Listing 2.26).

```
1 if($var == 'abc') { }         // safe
2 if($var === 'abc') { }        // safe
3 if(!strcmp($var, 'abc')) { }  // safe
4 if($var == 1) { }             // unsafe
5 if($var === 1) { }            // safe
```

Listing 2.26: Validating a variable's string content.

Caution is advised when the *equal* operator is used (`==`, line 4). This operator performs a type unsafe comparison by applying duck typing on its operands. Therefore, any string starting with the number *1* is typecasted to the integer *1* when compared with an integer. Thus, malicious characters in this string bypass the comparison to *1*. A type safe comparison is performed with the *identical* operator (`===`).

Explicit Whitelisting In order to compare input against a set of whitelisted values, an array can be used as lookup table, as shown in Listing 2.27. The lookup can be performed either by a *key* (line 2–4) or by a *value*.

```
1 $whitelist = array('a' => true,'b' => true,'c' => true);
2 if(isset($whitelist[$var])) { }              // safe
3 if($whitelist[$var]) { }                     // safe
4 if(array_key_exists($var, $whitelist)) { }   // safe
5 if(in_array($var, array('a','b','c'))) { }   // safe
6 if(in_array($var, array(1, 2, 3))) { }       // unsafe
7 if(in_array($var, array(1, 2, 3), true)) { } // safe
```

Listing 2.27: Using an explicit whitelist for validation.

Looking up a value in an array applies to the same rules than comparing two values with the *equal* operator. Thus, the example in line 6 is unsafe because the string *1abc* is typecasted to *1* and found successfully in the array. To avoid this, the *strict* parameter has to be set to *true*. Similar pitfalls occur when using the built-in function `array_search()`.

Implicit Whitelisting Next to an array, a value can be compared against a fixed set of items. For example, method and property names are limited to an alphanumerical character set. If a value matches one of these names, for example enforced with the built-in function `method_exists()`, it implies that no malicious characters are present.

Second-order Validation Similar to a whitelist, a value can be looked up in a resource, such as the file system or a database. Listing 2.28 shows an example where an email is looked up in the table *user*. The path in line 3 is only reached when a user with that email exists. Similarly, three additional examples show a check for the presence of a file name.

```
1 $var = addslashes($var);
2 $r = mysql_query("SELECT * FROM user WHERE mail='$var'");
3 if(mysql_num_rows($r)) { }
4 if(file_exists($var)) { }
5 if(realpath($var)) { }
6 if(stat($var)) { }
```

Listing 2.28: Database and file name lookup.

The safety of the validation depends on the present values in the database or the available file names. If the application allows to insert arbitrary email addresses to the database or to upload arbitrary file names, then the validation is unsafe [31].

2.3.4 Context-Sensitive Input Validation

Input validation can also be performed context-sensitively: for a subset of vulnerability types the data is validated against a safe character set or against the absence of malicious characters regarding the vulnerability type and markup context in a specific code path. Another vulnerability type or another markup context within the same path may still be exploitable. In the following, we introduce examples of context-sensitive input validation we encountered in practice.

Searching For a specific context, user input can be validated by proving the absence of a malicious character required for exploitation. For example, if no < character is found in the input, it can be considered safe regarding to XSS in the context of an HTML tag. Two typical searches are shown in Listing 2.29.

```
1 if(!strpos($var, '<')) { }           // unsafe
2 if(strpos($var, '<') === FALSE) { }  // safe
```

Listing 2.29: Searching for a specific malicious character.

The first example is unsafe because `strpos()` returns the offset at which the character was found in the string. If the string starts with a < character, offset *0* is returned that evaluates to *false* in the `if`-condition. Thus, the first validation can be bypassed.

Length Validation A specific string length can or cannot prevent exploitation, depending on the vulnerability type and its markup context. For example, in MySQL the SQL injection of the three characters `'-'` is equal to a `'or'1'='1` injection. For an XSS vulnerability, only three characters are not enough for exploitation. Thus, a string length validation, as the one shown in Listing 2.30, is context-sensitive.

```
1 if(strlen($var) < 3) { }
```

Listing 2.30: Validating the length of a variable.

Regular Expressions Regular expressions are an useful tool in order to perform very precise input validation. In Listing 2.31, three different examples are shown where only alphanumerical characters are allowed in the upfollowing path.

```
1 if(!preg_match('/[^\w]/', $var)) { }  // safe
2 if(preg_match('/\w+/', $var)) { }     // unsafe
3 if(preg_match('/^\w+$/', $var)) { }   // safe
```

Listing 2.31: Validating the character set with regex.

The first example ensures that no characters are present except for alphanumerical (`\w`) characters. The second example checks that alphanumerical characters are present. However, it fails to check the complete string range due to the missing boundary checks (compare to line 3). Hence, one alphanumerical character, at any position of the string, is enough to bypass the validation. More pitfalls regarding regular expressions can be found in Section 2.3.2.

2.3.5 Path Sensitivity

A security mechanism can also be spread across multiple paths of the control flow. In this case, path-insensitive code analysis reports false positives when impossible path combinations are considered [159]. In the following, we present examples for path-sensitive applications of security mechanisms and outline the challenges for static code analysis.

Path-sensitive Sanitization In Listing 2.32, the variable `$var` is implicitly sanitized. Initially, a check validates against a numerical data type. If this condition does not hold, the variable is then sanitized. For example, in line 2 the variable is set to the integer `0` which defuses it for the code after the `if`-block. Similarly, the variable can be unset (line 3) or a context-sensitive sanitization can be applied (line 4).

```
1 if(!is_numeric($var)) {
2    $var = 0;
3    //unset($var);
4    //$var = addslashes($var);
5 }
```

Listing 2.32: Path-sensitive sanitization.

Typically, static code analysis tools fail to recognize this kind of input sanitization because all execution paths are considered separately. Thus, when analyzing the execution path that skips the `if`-block, no variable modification is detected. However, this implies that the variable's value is already numerical.

Path-sensitive Termination A similar confusion of static analysis can occur when the program is terminated based upon input validation. In Listing 2.33, the program execution is halted if `$var` is not numerical. Alternatively, a loop can be aborted (`break`) or the control-flow of a user-defined function can be returned (`return`).

```
1 if(!is_numeric($var)) {
2    die("not numeric.");
3 }
```

Listing 2.33: Path-sensitive program termination.

A static analysis tool should not only be aware of the fact that there is no jump from the `if`-block to the following code, but also of the fact that the conditional termination of the program prevents any non-numerical characters after the `if`-block. Considering more complex code and the *halting problem* [142], which proves the undecidability of all program halts with another program, it is evident that static code analysis cannot correctly reason about all security mechanisms.

Path-sensitive Validation Another challenge for static analysis is path-sensitive usage of input validation. A typical example is given in Listing 2.34 where the variable `$error` is used to flag bad input.

```
1 if(!is_numeric($var)) {
2    $error = true;
3 }
4 if(!$error) { }
```

Listing 2.34: Path-sensitive validation.

The variable `$error` is independent from the variable `$var` which is analyzed for tainted input. Thus, its relevance for input validation is likely to be missed by path-insensitive static analysis. Contrarily, the analysis of all variables in all conditions of an execution path for input validation is very expensive for long paths and inter-procedural data flow.

2.4 Second-order Vulnerabilities and Multi-step Exploits

One common assumption underlying many detection and prevention approaches is that data that is already stored on the server is safe. However, an attacker might be able to bypass the defenses via so called *second-order* vulnerabilities if she manages to first abuse the web application to store the attack payload on the web server, and later on use this payload in a security-critical operation. Such vulnerabilities are often overlooked, although in practice they can have a severe impact. For example, XSS attacks that target the application's users are worse if the payload is stored in a shared resource and then distributed to all the users. Additionally, within *multi-step exploits* a vulnerability can be escalated to a more severe vulnerability. Thus, detecting second-order vulnerabilities is crucial for improving the security of web applications.

In this section, we introduce the nature of second-order vulnerabilities and multi-step exploits. First, we examine data flow through *persistent data stores* and the difficulties of analyzing such flows statically. We then present two types of second-order vulnerabilities as examples and demonstrate multi-step exploit attacks.

2.4.1 Persistent Data Stores

We define *persistent data stores* (PDS) as memory locations used by an application in order to store data. This data is available after the incoming request was parsed and can be accessed later on by the same application in order to reuse the data. The term *persistent* refers to the fact that data is stored on the web server's hard drive, although it can be frequently deleted or updated. Our definition also includes session data since information about a user's session is stored on the server and can be reused by an adversary. We now introduce three PDS commonly used by web applications.

Databases Databases are the most popular form of PDS found in today's web applications. A database server typically maintains several databases that consist of multiple tables. A table is structured in columns that have a specific data type and length associated with them. Stored data is accessed via SQL queries that allow to filter, sort, or intersect data on retrieval. In PHP, an API for database interaction is bundled as a PHP extension that provides several built-in functions for the database connection, and the query and access of data.

In contrast to other types of PDS, *writing* and *reading* to a memory location is performed through the same built-in query function. SQL has different syntactical forms of writing data to a table. Listing 2.35 shows three different ways in SQL to perform the same query. While the first two queries explicitly define the column names, the third query does not. We refer to the first type as *specified write* and to the second type as *unspecified write*.

```
1 // specified write
2 INSERT INTO users (id,name,pass) VALUES (1,'admin','foo')
3 INSERT INTO users SET id = 1, name = 'admin', pass = 'foo'
4 // unspecified write
5 INSERT INTO users VALUES (1,'admin','foo')
```

Listing 2.35: Writing to the database table *users* in SQL.

Both types convey a difficulty for static analysis of the query: a *specified write* reveals the column names where data is written to, but does not reveal if there are any other columns in the table that are filled with default values. This hinders the reconstruction of table structures when statically analyzing SQL queries of an application. An *unspecified write* tells us exactly how many columns exist, but does not reveal its names. When the columns are later on accessed by name, it is unclear which column was filled with which value. The same principle applies for read operations. A *specified read* reveals the accessed column names in a field list, whereas an *unspecified read*, indicated by an *asterisk* character, selects all available columns without naming them explicitly.

In PHP, the *queried* data is stored as a result resource. There are different ways to fetch the data from the result resource with built-in functions, as shown in Listing 2.36.

```
1 // numeric fetch
2 $row = mysql_fetch_row($res);      echo $row[1];
3 // associative fetch
4 $row = mysql_fetch_assoc($res);    echo $row["name"];
5 $row = mysql_fetch_object($res);   echo $row->name;
```

Listing 2.36: Fetching data from a database result resource.

Basically, *numerical* and *associative fetch* operations are available. The first method stores the data in a numerically indexed array whereat the index refers to the order of the selected columns. The *associative fetch* stores the data in an array indexed by the columns' names. It is also possible to store the data in an object where the property names equals the column names. The key difference is that the *associative fetch* reveals the accessed column names while the *numerical fetch* does not.

All different combinations of writing, reading, and accessing data can occur within a web application. In certain combinations, it is not clear which columns are accessed without knowledge about the database schema, e. g., in case that data is written *unspecified* and fetched associatively. In practice, however, we are often able to reconstruct the database schema by parsing `CREATE TABLE` statements within the source code or in *.sql* files.

Session Data A common way of dealing with the state-less HTTP protocol are *sessions*. In PHP, the `$_SESSION` array provides an abstract way of handling session data that is stored within files (default) or databases. A session value is associated with an alphanumerical key that represents the memory location. Listing 2.37 shows how a *name* provided by a user is stored within the session key *username*. Later on, the name is retrieved again by accessing this session key. Note that the `$_SESSION` array needs to be treated like any other superglobal array in PHP and it can be accessed in any context of the application. As any other array, it can be dynamically and inter-procedurally accessed or modified, and it can have multiple key names (see Section 2.1). Besides the `$_SESSION` array and the deprecated `$HTTP_SESSION_VARS` array, the built-in functions `session_register()` and `session_decode()` can be used in order to set session data.

```
1 // set
2 $_SESSION['username'] = $_GET['name'];
3 // get
4 $name = $_SESSION['username'];
```

Listing 2.37: Setting and getting a session variable.

File Names A common source for vulnerabilities is an unsanitized file name. Developers often overlook that the file name of an uploaded file can contain malicious characters and thus can be used as a PDS for an attack payload. For example, Unix file systems allow any special characters within file names, except for the slash and the null byte [69]. NTFS allows characters, such as the single quote that can be used for exploitation [85]. For detecting second-order vulnerabilities, we need to determine paths where files with arbitrary names are located. The analysis of a file upload reveals the path a file is written to and if the file is named as specified by the user. In PHP, a file that is submitted via a multi-part POST request is stored in a temporary directory with a temporary file name. The temporary and the original file name is accessible in the superglobal `$_FILES` array. Furthermore, built-in functions, such as `rename()` and `copy()`, can be used by an application to rename a file on the server. The code in Listing 2.38 copies an uploaded file with its submitted file name to the *uploads/* directory. The directory is later on accessed and each file name is printed. Note that also directory names can be used as PDS, for example when created with the built-in function `mkdir()`.

```
1 define('UPLOAD_DIR', 'uploads/');
2
3 move_uploaded_file($_FILES['file']['tmp_name'], UPLOAD_DIR. $_FILES['file']['name']);
4
5 $files = scandir(UPLOAD_DIR);
6 foreach($files as $file) {
7    echo $file . "<br />";
8 }
```

Listing 2.38: File upload and file name manipulation.

Other PDS Additionally, other less popular PDS exist. For example, data can be retrieved from a CGI environment variable, a configuration file, or an external resource, such as an FTP or SMTP server [17]. However, these PDS are rarely used in practice and decisions can only be made with preconfigured whitelists. We only consider PDS that are tainted by the application itself and not through a different channel. Analyzing the data flow through file content could be an interesting addition in the future. Here, the challenge is to determine the part of a given file that data is written to and from that data is read because the structure of the data within the file is unknown.

Note that data stored via PHP's built-in functions `ini_set()` or `putenv()` only persists for the duration of the current request. At the end of the request, the environment is restored to its original state. Thus, they do not hold to our definition of a PDS.

2.4.2 Second-order Vulnerabilities

To recall, a (*first-order*) taint-style vulnerability occurs if data controlled by an attacker is used in a security-critical operation. In the data flow model, this corresponds to tainted data literally flowing into a sensitive sink within one possible data flow of the application. We classify a *second-order* vulnerability as a taint-style vulnerability where the data flows through one or more PDS. Here, the attack payload is first stored in a PDS and is later retrieved and used in a sensitive sink. Thus, *two* distinct data flows require analysis: (i) source to PDS, and (ii) PDS to sink.

In the following, we introduce two examples where the source is stored in a PDS before it reaches the sensitive sink. In general, every combination of a source, sensitive sink, and a PDS is possible. Depending on the application's design, the flow of malicious data occurs within a single or multiple attack requests (e. g., when different requests for *writing* and *reading* are required).

Persistent Cross-Site Scripting As introduced in Section 2.2.1, XSS occurs when unsanitized user input is reflected to the HTML page and an attacker can inject malicious HTML or JavaScript code into the response. We speak of *persistent cross-Site scripting* if the attacker's payload is stored in a PDS first, read by the application again, and then printed to the response page. In contrast to *non-persistent* (*reflected*) XSS, the attacker does not have to craft a suspicious link and send it to a victim. Instead, all users of the application who visit the affected page are automatically attacked, making the vulnerability more severe. On top of that, a *persistent XSS* vulnerability can be abused to spread an XSS worm [80, 125].

Listing 2.39 depicts an example of a *persistent XSS* vulnerability. The simplified code allows to submit a new comment which is stored in the table `comments` together with the name of the author. If no new comment is submitted, the code lists all previously submitted comments that are fetched from the database. While the comment itself is sanitized in line 7 with the built-in function `htmlentities()` that encodes HTML control characters, the author's name is not sanitized in line 6 and is thus affected by XSS. Note that if the source code is analyzed top-down, it is unknown at the point of the `SELECT` query if malicious data can be inserted into the table `comments` by an adversary, or not.

```
1  if(empty($_POST['submit'])) {
2     // list comments
3     $res = mysql_query("SELECT author,text FROM comments");
4     foreach(mysql_fetch_row($res) as $row) {
5        $comment = mysql_fetch_array($row);
6        echo $comment['author'] . ': ' .
7        htmlentities($comment['text']) . "<br />";
8     }
9  }
10 else {
11    // add comment
12    $author = addslashes($_POST['name']);
13    $text = addslashes($_POST['comment']);
14    mysql_query("INSERT INTO comments (author, text) VALUES ('$author', '$text')");
15 }
```

Listing 2.39: Example for *second-order XSS* vulnerability.

Second-Order SQL Injection A SQLi vulnerability occurs when a web application dynamically generates a SQL query with unsanitized user input (see Section 2.2.2). In Listing 2.40, a second-order SQL injection is shown where the attacker's payload is stored in a PDS. In the following, we analyze the vulnerable code step by step. Starting in line 7, user-supplied credentials are checked for authentication. If the credentials are valid, the session key `loggedin` is set to *true* and the user-supplied name is saved into the session key `user`. In case the user-supplied data is invalid, the failed login attempt is logged to

the database with the help of the user-defined `log()` function (line 1–5). At this point, a *second-order SQL injection* occurs: if an attacker registers with a malicious username, this name is written to the session key `user` and after a second failed login attempt, it is used in the logging SQL query.

```
1  function log($error) {
2     $user = $_SESSION['user'];
3     $query = "INSERT INTO logs (error, user) VALUES ('$error', '$user')";
4     mysql_query($query);
5  }
6
7  if(validAuth($_POST['user'], $_POST['pass']) {
8     $_SESSION['loggedIn'] = true;
9     $_SESSION['user']     = $_POST['user'];
10 }
11 else {
12    log('Failed login attempt');
13 }
```

Listing 2.40: Example for a *second-order SQLi* vulnerability.

2.4.3 Multi-Step Exploits

We introduce multi-step exploits as a subclass of second-order vulnerabilities. Within a second-order vulnerability, the first order (e. g., safe writing of user input into the database or a file path) is not a vulnerability by itself. However, *unsafe* writing can lead to other vulnerabilities. We define a multi-step exploit as the exploitation of a vulnerability in the second order that requires the exploitation of an unsafe writing in the first order. Thus, a multi-step exploit is a subclass of a second-order vulnerability and it can drastically raise the severity of the first vulnerability. Since we only consider databases, sessions, and file names as PDS in our analysis, the following three vulnerabilities are relevant:

- A *SQLi* in an `INSERT` or `UPDATE` statement allows full compromise of all columns in the specified table. An attacker can write a payload in all columns of the table that the application reads from at a later point. Furthermore, a SQLi in a `SELECT` query allows an attacker to return arbitrary data by injecting an `UNION SELECT` statement with arbitrary values (see Section 2.2.2).

- A *path traversal* vulnerability allows to change the current directory of a file operation to another location. Arbitrary file names can be created in arbitrary locations if a *path traversal* vulnerability affects the naming or creation of files. Therefore, this vulnerability can be used to bypass a validation that bases upon the existance of a specific file name (see Section 2.3.3).

- An *arbitrary file write* vulnerability can modify or create a new session file, leading to the compromise of all session values. Accordingly, data received from a file and then used in a sensitive sink can now be considered as tainted.

2.5 PHP Object Injection

Memory corruption vulnerabilities, such as buffer overflows, format string bugs, and dangling pointers, are known for a long time and still constitute an intractable class of programming mistakes [129,145]. While defense techniques, such as *address space layout randomization* (ASLR) and *data execution prevention* (DEP), are widely deployed to hamper the exploitation of such vulnerabilities, an adversary can still utilize different techniques to circumvent such defenses. Especially *code reuse* techniques, such as for example *return-to-libc* [118], *return-oriented programming* (ROP) [104], and *jump-oriented programming* (JOP) [15], have received a lot of attention since they are able to bypass several kinds of security protections. With ROP and JOP, an attacker reuses available code fragments in memory (so called *gadgets*) and joins them together to construct the attack payload piece by piece (so called *gadget chains*) in scenarios where she cannot inject her own code.

In 2009, Esser showed that code reuse attacks are also viable in PHP-based web applications [41,42]. A *PHP Object Injection* (POI) vulnerability occurs when unsanitized user input is used during the deserialization of data in a given application. PHP features so called *serialization* and *deserialization* functions that allow a programmer to store data of any type in an unified string format. This format makes it easy to transfer combined data structures and is often misused to create multidimensional cookies and similar data structures.

Since PHP allows deserialization of arbitrary objects, an attacker could be able to inject a specially prepared object with an arbitrary set of properties into the application's scope. Depending on the context, the attacker can trigger so called *magic methods* [133] and this potentially leads to a variety of vulnerabilities. Note that the type of vulnerability is highly dependent on the classes' implementation of their *magic methods*. Each *magic method* might call another (potentially security-relevant) PHP function (e.g., `eval()` or `fwrite()`) with attacker-controlled member variables as arguments which can lead to remote code execution, file inclusion, SQL injection, and any other kinds of vulnerabilities.

In this section, we first introduce the concepts of *magic methods* (Section 2.5.1) and *serialization* (Section 2.5.2) in PHP. Both features form the basis to exploit a POI vulnerability by utilizing *Property Oriented Programming*. This exploit technique combines both features and is described in Section 2.5.3. It is one of the most sophisticated attack techniques against PHP applications since it requires reusing already existing code in the application's classes.

2.5.1 Magic Methods in PHP

The concept of *object-oriented programming* (OOP) was considerably enhanced in version 5 of PHP and since then includes destructors, exceptions, interfaces, and further object-oriented concepts. OOP allows to logically encapsulate data and functionality in *objects*, while their implementation resides in the *class* definition. Each class can be initialized into an object that contains properties and methods that are defined in their designated class. These properties are called *attributes* (or *fields*), while a *method* describes a function that is accessible to an object.

Magic methods play an important role when exploiting POI vulnerabilities since they are *automatically* executed upon specific events. As we will see later on, they can be used to *start* a POP gadget chain. The following magic methods fulfill a special purpose and can be defined once per class:

- `__construct()`: This magic method implements the constructor inside a class that is called whenever a new object of that class is created. It is often used to initialize the object's attributes or to run other code before the object can actually be used.

- `__destruct()`: In contrast to the `__construct()` function, `__destruct()` is executed whenever the script terminates or when the reference count of an object reaches zero. It is often used to invoke code that cleans up used data or that terminates connections possibly established after the object was created.

- `__call()`: This function is always invoked when an inaccessible method of an object is called (e. g., `$obj->invalid_method()`). It is useful in terms of error handling, since accessing invalid methods usually results in a fatal error and termination of the PHP application.

- `__callStatic()`: Similarly to the `__call()` method, this magic method catches inaccessible calls in a static context (e. g., `obj::invalid_method()`).

- `__get($name)`: The method `__get()` is automatically called when trying to *read* private, protected, or non-existent properties of an object. Since private and protected properties cannot be directly accessed outside of the object, the parameter `$name` is used to reference the desired property.

- `__set($name, $value)`: The method `__set()` is automatically called when trying to *write* to private or protected properties of an object. Because this is prohibited, this function allows the application to handle assignments, such as for example `$obj->private = 'value'`.

- `__isset()`: Similar to previously mentioned methods, this function is called whenever `isset()` or `empty()` are used on a non-existent property.

- `__unset()`: Every time `unset()` is used on non-existent properties, this function is called with an argument which describes the name of the variable that the application wants to unset.

- `__sleep()`: This magic method is triggered whenever an object is serialized. It gives the programmer the ability to let the object run any sort of cleanup-code before serialization.

- `__wakeup()`: In contrast to the method `__sleep()`, `__wakeup()` is called directly after deserialization. It is often used to reinitialize the application's state that was lost during serialization, for example the connection to a database.

- `__toString()`: Whenever an object is used in a string context (e. g., when it is concatenated with a string), then this method is invoked in order to return a string representation of the object.
- `__invoke()`: This method is called whenever an object is used as a dynamic function name (e. g., `$obj()`).
- `__set_state($properties)`: Within an application, the function `var_export()` is used in order to display any sort of data as parsable PHP code. If an object is used as argument, the method `__set_state()` is called in order to define which of the properties are exported.
- `__clone()` This function is called when an object is cloned by the `clone` operator. It is equivalent to *copy-constructors* known in other languages. By implementing this method in a class, the programmer can specify what exactly should happen during cloning.

2.5.2 Serialization in PHP

PHP supports the serialization and deserialization of all defined data types—including objects. Serialization is realized through the built-in function `serialize()` which accepts a single parameter and returns a serialized string that can be fed into `unserialize()` in order to retrieve said data again. This string is represented in an unified format which consists of several identifiers that specify the serialized data type. These identifiers have the following purpose:

- `a:` – defines that the passed parameter is an array. `a:` is always followed by a numerical value which specifies the size of the array.
- `i:` – simply defines a numerical value, e. g., `i:8;`.
- `b:` – specifies a boolean value, e. g., `b:0;` or `b:1;`.
- `s:` – defines a constant string. `s:` is always followed by a numerical value which declares the length of the string, e. g., `s:4:"test";`.
- `S:` – defines a constant string in an encoded format.
- `O:` – represents an object in its serialized form. `O:` is followed by the length of the class name and by the name itself, e. g., `O:1:"A"`. It is then followed by the number of properties and the defined properties themselves. Note that a property can also consist of another object with its defined properties.

Further identifiers, such as `r:` and `R:`, exist and can be used to store references, but they are out of scope for an attack. An example of the functionality behind PHP's serialization is given in Listing 2.41. Line 2 serializes the array defined in line 1 and it therefore returns the string in line 4 which is then fed into `unserialize()` again. Line 6 shows that the deserialization of the array returns the same values in the same structure as they were previously defined.

```
1 $arr = array(1 => 2, 3 => "string");
2 $serialized = serialize($arr);
3 print $serialized . "\n";
4 // a:2:{i:1;i:2;i:3;s:6:"string";}
5 var_export(unserialize($serialized));
6 // array ( 1 => 2, 3 => 'string' )
```

Listing 2.41: Exemplary serialization of an array.

2.5.3 Property Oriented Programming

Property Oriented Programming (POP) abuses the ability of an attacker to arbitrarily modify the properties of an object which is injected into a given web application. Thus, the data and control flow of the application can be manipulated. There are two preconditions that a PHP application needs to meet so that POP can be used to exploit a POI vulnerability. First, at least one *magic method* that is called during the application's runtime needs to be defined in an object's class which the attacker wants to inject. Second, the chosen class needs to be loaded within the scope of the vulnerable `unserialize()` call that the attacker passes her input into.

Each magic method can either be *context-dependent* or *context-independent*. *Context-dependent* means that an object has to be used in a certain way so that a magic method is invoked (see Section 2.5.1). Other magic methods are called *automatically* during the application's lifetime: the method `__wakeup()` and `__destruct()` is *context-independent* since `__wakeup()` is always called directly after deserialization of an object and `__destruct()` is always called once the object is destroyed. Both methods might operate on properties that can be arbitrarily defined when the object is deserialized.

Passing user input into the `unserialize()` function enables an attacker to inject specially crafted objects with chosen properties that will be used inside of the *magic method*. However, when only context-dependent methods exist, such as `__toString()` or `__call()`, the attacker has to choose a code path where the deserialized object is used accordingly to trigger the *magic method*. These code paths are often a lot more scarce and thus context-independent methods are a better choice for these type of attacks.

Each *magic method* can also call different methods of other objects which are linked to the first object as members. In this scenario, it is recommended to examine all other object methods, which can also be denoted as *gadgets*, for dangerous sinks that can all be joined to a complete injectable POP chain. Listing 2.42 shows an excerpt of a vulnerable application where three *gadgets* are combined in order to achieve an arbitrary file deletion.

The POI vulnerability occurs in line 19, where user input is deserialized. Note that an application often does not intend to deserialize objects but rather arrays. By forging a cookie with the content shown in lines 20–21, the attacker injects a `Database` object with the `$handle`-property set to a `TempFile` object. Its `$filename` property is then set to the `../../.htaccess` file the attacker attempts to delete. When the application terminates, the injected `Database` object will automatically execute its destructor (line 15). The destructor will then use the `$handle`-property to execute its `shutdown()` function. Because the attacker loaded the class `TempFile` into this property, the function `shutdown()` of `TempFile` is triggered. It inherits this method from the `File` class in line 2.

```
1  class File {
2     public function shutdown() {
3        $this->close();
4     }
5     public function close() {
6        fclose($this->h); // harmless
7     }
8  }
9  class TempFile extends File {
10    public function close() {
11       unlink('/var/www/tmp/logs/' . $this->filename); // !!
12    }
13 }
14 class Database {
15    public function __destruct() {
16       $this->handle->shutdown();
17    }
18 }
19 $data = unserialize($_COOKIE['data']);
20 //  O:8:"Database":1:{s:6:"handle";
21 //  O:8:"TempFile":1:{s:8:"filename";s:15:"../../.htaccess";}}
```

Listing 2.42: Exploitation of a POI vulnerability.

Next, the method `shutdown()` invokes the method `close()`. Although this method is harmless in the `File` class, it is overwritten in the class `TempFile` with a harmful method that deletes the specified `.htaccess` file (line 11).

Note that an *initial gadget* (in this case `Database`'s destructor) is required in order to begin an execution flow of already existing code, defined in the object's methods. For every set of objects multiple variations of *gadgets* can be combined, each leading to another class of vulnerability. As manually checking the application's source for useful *gadgets* is cumbersome and time consuming, an automated approach is desirable.

2.6 Discussion

In this chapter, we introduced the variety of features provided by the popular PHP language. Its highly dynamic language features and rich set of built-in functions allow to quickly develop new applications and hint at the reason of the PHP language's great adoption in the World Wide Web. We introduced various sources for user input, dynamic language constructs, security sensitive operations, and security mechanisms for protection. Consequently, we also presented the theoretical and technical background of different vulnerability types that can occur when no security mechanism is in place or is misplaced. More specifically, we presented intricacies and pitfalls that can lead to subtle security bugs which are hard to spot in code reviews and hint at the reason for the high amount of security vulnerabilities found in PHP applications. Recently, we are seeing more and more complex vulnerabilities reported, such as second-order and POI vulnerabilities, as well as a growing trend of exploitation by attackers. Hence, reliable detection and prevention is becoming more and more important.

For the automated detection of these vulnerabilities, an important recollection is that all taint-style vulnerabilities follow one principle: unsanitized or unvalidated user input literally *flows* into a security sensitive operation of the application and allows malicious

users to change the behavior and the actions of the operation. Static code analysis can detect these issues by marking user input as *tainted* sources and report a vulnerability if a source is used unsanitized in a sensitive sink. The challenge is to model the data flow through the dynamic language features and to evaluate the correctness of all applied security mechanisms in order to avoid false alarms. Moreover, the vast amount of built-in functions in PHP is two-edged for the application of static analysis in practice. On the one hand, it requires an intense configuration of the static analysis engine. On the other hand, then, the analyses of called built-in functions lead to more precise results than the analyses of several custom user-defined functions with the same purpose in other programming languages would. We also introduced security issues that depend on the PHP version or PHP configuration. Additionally, it is evident that complex vulnerability types, such as second-order or PHP object injection vulnerabilities, require a more comprehensive analysis. In the next chapter, we propose a novel design of such static code analysis tool and introduce our approach to these challenges.

Chapter 3

Designing a Static Code Analysis Tool

The manual detection of all subtle security vulnerabilities in hundreds of thousand lines of code written in an intricate scripting language can be incomplete and inefficient. In this chapter, we introduce our novel static analysis approach for the *automated* security analysis of modern PHP applications in order to detect complex security vulnerabilities. Our goal is to help users minimize the time and costs of a manual review and to convey security expertise to the user in a limited degree. Although we focus on the PHP language, generalizing our approach to different languages is possible by applying our analysis algorithms to its language features. This chapter is structured as follows.

In Section 3.1, we first present related work in the field of static and dynamic security analysis of PHP applications and reveal their common weaknesses. Then, an overview of our approach is presented in Section 3.2. Here, we distinguish previous analysis principles from our novel techniques and outline the rest of this chapter in detail. From a highlevel perspective, our analysis can be devided into five analysis steps, as illustrated in Figure 3.1. The source code of an application (a) is first transformed into an abstract representation (b) which is then split into several *basic blocks* (c). These blocks are separately analyzed (see Section 3.4) and connected with *block edges* (see Section 3.5) to a *control flow graph* (d). Its construction algorithm is explained in Section 3.3. Based on this graph, we automate the process of finding security vulnerabilities by using a backwards-directed *taint analysis* (depicted as red arrows in Figure 3.1 (d), details in Section 3.7). The analysis is able to evaluate *second-order* vulnerabilities (see Section 3.8) and to generate POP chains (see Section 3.9). The limitations of our approach are discussed in Section 3.10.

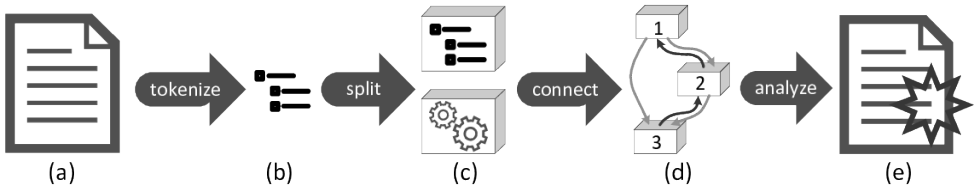

Figure 3.1: A highlevel overview of our design approach. The novelty is in the last three steps: a precise language feature analysis (c), a highly efficient data flow analysis (d), and a wide variety of supported vulnerability types (e).

3.1 Related Work

Web applications are widely used in the modern Web and as a result, security analysis of such applications has attracted a considerable amount of research. Due to its widespread usage, specifically the PHP language has received a lot of attention over the last years. In the following, we discuss work related to static and dynamic taint analysis of PHP code and clarify how we advance the state-of-the-art. Furthermore, prior work on string and security mechanism analysis is related to our approach and we discuss this area as well.

3.1.1 Static Analysis

Static analysis approaches analyze the application's source code without execution. Huang et al. developed a static code analysis tool for PHP called *WebSSARI* [61] based on a CQual-like [43, 44] type system. Compared to our approach, it has certain limitations. First of all, it works only *intra-procedural* and not *inter-procedural*. That means that WebSSARI is able to handle the program flow through user-defined functions, but it does not consider the context from where the function is called. Second, it does not handle dynamic features of PHP, such as dynamic arrays or dynamic includes, which implies that many vulnerabilities will be missed for large PHP applications. In a follow-up paper, Huang et al. presented a related approach based on bounded model checking [60] which has similar limitations.

Xie and Aiken presented a static analysis algorithm for detecting SQL injection vulnerabilities in PHP applications using *block* and *function summaries* [154]. It is both intra-procedural and inter-procedural and handles more dynamic features of PHP. However, we found several limitations in their approach. For example, it does not handle multi-dimensional arrays or object-oriented code and thus misses vulnerabilities in modern PHP applications. Their approach for file inclusion is error prone and can lead to path explosion. Additionally, their implementation only supports SQLi vulnerabilities in a context-insensitive way and does not model built-in functions. In contrast, our approach supports object-oriented code, covers 36 vulnerability types context-sensitively, and precisely models built-in functions. These three features enable us to detect new vulnerabilities in modern PHP applications. The direct comparison of our approach with the approach by Xie and Aiken discussed in Section 4.1.6 demonstrates that our approach outperforms their method.

Jovanovic et al. developed *Pixy*, an open source, static code analyzer for PHP written in Java [66, 68]. A lot of work has been put into modeling aliases which are supported by our tool only in a limited way. However, we found only very few recent PHP applications actually using aliases and only in a rather simple manner. The down-side of Pixy is that it only supports XSS and SQLi vulnerabilities, does not support object-oriented code, and has only 29 built-in functions configured which leads to false negatives. False positives occur due to missing or imprecise modeling of file inclusions, built-in functions, and markup context analysis. The direct comparison in Section 4.1.6 demonstrates that our approach outperforms Pixy with significantly less false negatives and positives.

There are static analysis approaches which target other classes of security vulnerabilities. For example, *SaferPHP* [121] attempts to find *semantic attacks* (e. g., denial of

service attacks due to infinite loops caused by malicious inputs, or unauthorized database operations due to missing security checks) within web applications. *RoleCast* [119] identifies security-critical variables and applies role-specific variable consistency analysis in order to identify missing security checks, while *PHANTM* [74] detects type errors in PHP code. Such kinds of software defects are out of scope for our analysis.

Furthermore, we are not aware of any plain *static* code analysis implementation handling second-order vulnerabilities. Previous approaches are not able to decide whether data fetched from persistent stores is tainted or not. Assuming all data is tainted would lead to a high number of false positives, while a conservative approach can miss vulnerabilities. Also, no existing analysis tool is capable of deciding whether a given POI vulnerability is actually exploitable or not. This is a challenging analysis task since we need to identify a combination of gadgets in the code that allow an attacker to trigger another vulnerability by manipulating the control and data flow. Additionally, complex OOP features of PHP require a comprehensive analysis and—to the best of our knowledge—no existing static analysis tool for PHP-based web applications supports scalable OOP analysis.

However, the challenges we address in order to perform an efficient OOP analysis on large applications were previously addressed in other programming languages. For example, several static code analysis approaches have been proposed to perform points-to analysis for the weakly typed JavaScript language [7, 48, 49, 62, 123]. A broad overview on different approaches to perform object-sensitive analysis was performed by Smaragdakis et al. [116]. They introduce type-sensitive analysis as a more scalable solution that picks its context based on types instead of objects. Livshits and Lam proposed a static analysis approach to detect security vulnerabilities in Java applications [81]. Tripp et al. designed static taint analysis for Java and implemented their approach in the *TAJ* system [139]. In general, these approaches cannot be adopted to the PHP language due to missing type information in PHP.

3.1.2 Static Security Mechanism Analysis

A variety of static analysis approaches have been proposed to automatically identify security vulnerabilities in PHP applications based on insufficient sanitization and validation. Sanitization analysis was also applied to other programming languages, such as JavaScript [105, 106], Java [22], and ASP.NET [58, 79, 107].

Zheng and Zhang introduced path-sensitive static analysis for PHP applications with *Z3-str* [159]. They leverage a modified version of the *Z3* SMT solver that is also capable of analyzing strings. Shar and Tan proposed static code attributes for predicting SQLi and XSS vulnerabilities [113,114]. Among their attribute vectors are six validation and six sanitization mechanisms. Other security mechanisms introduced in Section 2.3 are missed and will likely lead to false positives.

Yu et al. built an automata-based string analysis tool called *STRANGER* [156] based on the static code analysis tool Pixy [66]. STRANGER detects security vulnerabilities in PHP applications by computing possible string values using a symbolic automata representation of common string functions, including escaping and replacement functions. Later, they automatically generated sanitization statements for detected vulnerabilities by using regular expression replacements [157]. Balzarotti et al. combine static and dynamic

analysis techniques to identify faulty custom sanitization routines [5]. The static analysis component of their tool called *Saner* extends Pixy and analyzes string modification with automata, while the dynamic component verifies analysis results to reduce false positives. The drawback of STRANGER and Saner is that they are only as good as their test-cases. Even if all configured attack patterns for one vulnerability type are filtered correctly, other attack patterns could exist that bypass the sanitization undetected. Additionally, none of these tools detect input validation, a security mechanism covered by our approach.

Wasserman and Su leverage string analysis with context free grammars in order to detect XSS [150] and SQLi [149] vulnerabilities based on insufficiently-checked untrusted data. They cover string replacement and escaping, while path-sensitive input validation leads to false positives. Minamide developed a string analyzer to approximate the output of PHP applications using a context-free grammar [87]. It models a variety of sanitization functions but can only prove the absence of predefined attack vectors.

3.1.3 Dynamic Analysis

When using a *dynamic analysis* approach, the application is executed with a specific input and its behavior is analyzed at runtime. Nguyen-Tuong et al. implemented a taint mode for the PHP interpreter that tracks tainted strings throughout the information flow [95] in a similar way to Perl's taint mode [2]. Monga et al. proposed the hybrid analysis framework *Phan* that monitors sensitive statements at runtime which are previously detected by static PHP bytecode analysis [89]. The PHP extension *Diglossia* [120], and others [51, 124, 155], aim to detect SQL injection attacks at the database layer. These approaches can only detect attacks which are triggered within a monitored execution path or query and, thus, suffer from false negatives. Accordingly, approaches with input generation were developed. Wasserman et al. proposes an algorithm for the automated test generation for PHP code by gathering constraints during symbolic execution and using values collected at runtime [151]. *Apollo* [4] combines symbolic and concrete execution techniques together with explicit-state model checking. The authors tested their tool with *phpBB2* version 2.0.21 and detected several vulnerabilities. Our prototype detected 13 novel vulnerabilities in version 2.0.23 of *phpBB2*, indicating that our approach is capable of discovering flaws not identified by Apollo (see Section 4.1). *Ardilla* [71] aims at detecting both SQL injection and XSS vulnerabilities by leveraging the input generator from Apollo. It symbolically tracks taint information through execution and automatically generates concrete exploits. Furthermore, several *black-box* approaches exist which test web applications with predefined sets of attack patterns and analyze the application's response [9, 11, 59, 84, 110].

3.2 General Overview

In this section, we first provide a general overview of our approach and our novel analysis techniques. We use *block*, *function*, and *file summaries* in order to store the results of data flow analysis within each unit and to built an abstract data flow model for efficient analysis [37, 154]. More precisely, the following steps are taken:

1. For each PHP file in the project, an *Abstract Syntax Tree* (AST) is built, based upon PHP's open source internals. Then, all user-defined functions are extracted and relevant information like the name and parameters are stored in the environment. The body of the function is saved as separate AST and is removed from the *main* AST of the parsed file.

2. Similarily to functions, we extract class definitions from the ASTs. For static classes, we collect predefined properties and class constants. During data flow analysis, access to this static content is inferred instantly. Moreover, we build a *class hierarchy* [34, 127] based on the inheritance of each class (e. g., `class A extends B`). To answer the questions *who extends whom* and *who is extended by whom*, it is built in both directions. All defined methods are stored in the analysis environment as user-defined functions, but are linked to their native class. Additionally, we extract type information of parameters whenever possible and we construct *method fingerprints* (see Section 3.4.6).

3. We start transforming each *main* AST into a *Control Flow Graph* (CFG). Whenever a node of the AST performs a conditional jump, a new basic block is created and connected to the previous basic block with a block edge. The jump condition is added to the block edge and the following AST nodes are added to the new basic block (see Section 3.3).

4. We simulate the data flow of each basic block as soon as a new basic block is created (see Section 3.4). For this purpose, we perform a lightweight heap analysis of our intermediate representation of data. The analysis results are integrated into the so called *block summary* that is created during simulation and sums up the data flow within a block.

5. If a call to a previously unknown user-defined function is encountered during simulation, the CFG is built from the function AST and a *function summary* is created once with intra-procedural analysis (see Section 3.6.1). Then, the pre- and post-conditions for this function can be extracted from the summary and inter-procedural analysis is performed at call-site (see Section 3.6.2).

6. We conduct a taint analysis beginning from the currently simulated basic block for each vulnerable parameter of a user-defined function or of a configured sensitive sink (see Section 3.7).

Furthermore, we perform the following novel analysis steps in order to refine our results:

- Next to data types, we track the sanitization status for different vulnerability types and the encoding status during data flow analysis. This allows us to model different sanitization methods throughout the data flow (see Section 3.4.1).

- We summarize object- and field-sensitive data flow analysis in order to analyze object-oriented PHP code. Our novel analysis technique allows us to maintain the efficient concept of data flow summaries by assisting the backwards-directed data flow analysis with a forwards-directed, object-oriented analysis (see Section 3.4.4).

- Instead of connecting CFGs of included files into the current CFG, we model included files as functions. This prevents a redundant analysis of included files and it shortens the analysis paths of the CFG (see Section 3.4.6).
- We model a total of 1 243 built-in functions to recognize a variety of data flows, sanitizations, validations, encodings, sources, and sinks (see Section 3.4.7). This step is critical to perform a comprehensive security analysis.
- We simulate block edges and summarize their validation effects (see Section 3.5). This allows us to detect input validation mechanisms for specific control flow paths.
- Our string and taint analysis is performed backwards-directed (see Section 3.7). Intermediate results gathered from the block summaries are cached for each basic block which enables a highly performant analysis.
- We perform context-sensitive string analysis to refine our taint analysis results based on the current markup context, source type, and PHP configuration. Furthermore, it allows us to evaluate readings and writings of sensitive sinks to persistent data stores which enables us to detect second-order vulnerabilities (see Section 3.8).
- We invoke and summarize the analysis of magic methods whenever appropiate according to PHP's OOP language features. This enables the detection of POP gadget chains for POI vulnerabilities (see Section 3.9).

3.3 Control Flow Graph

After the application's source code is split into AST's for user-defined functions, methods, and files, a control flow graph is constructed for each unit. The `CFGBuilder` algorithm is initiated with the AST nodes of each *main* and *function AST*. It splits conditional program flow into linked basic blocks and initializes their simulation (covered in Section 3.4).

First, the `CFGBuilder` creates a new `BasicBlock` which is stored as `currentBlock`. Next, it loops through all root nodes of the AST and adds all nodes that are *not* a statement to the node list of the `currentBlock`.

If the control flow is deferred, a new `CFGBuilder` is initiated recursively for every branch that the statement introduces and the new `currentBlock` is linked to the previous `currentBlock`. Each constraint is added as first node of the basic block to ensure that it is part of the simulation process. For example, a variable declaration can occur within an `if`-constraint and must be part of the AST.

Loops are handled as one basic block. The loop constraint is analyzed and *looped* variables are identified, such as a repeatedly incremented variable within a `for`-statement. For these variables, all possible values are considered during data flow analysis, e. g., when used to access an array by key. While this may introduce imprecision, our evaluation shows that this approach is sufficient to detect vulnerabilities in real-world applications (see Section 4.1.3 for an example).

The `CFGBuilder` algorithm stops when the program flow is halted with a stop statement or when all statements are parsed and all subnodes are added to a basic block.

3.4 Simulating Basic Blocks

The simulation of a basic block is initiated during CFG construction whenever a statement occurs that splits the control flow into new basic blocks. Then we simulate the *current block* before we move on to the next basic block.

The purpose of the simulation is to create a summary of the data flow within *one* basic block of the CFG. In order to do so, we loop through all AST nodes of the basic block and parse assignments and function calls. These nodes can perform a data assignment whose symbolic value is stored in the *block summary*. The block-internal data flow is simulated by *static symbolic execution* (in the same sense as the analysis by Xie and Aiken [154]), while the data flow through PHP built-in functions is simulated by using fingerprints and *abstract interpretation* [26, 27]. Furthermore, we parse the `global`, `exit`, and `return` statements and add their effects to the block summary.

3.4.1 Intermediate Representation

Our language set for the intermediate representation of memory locations are *data symbols*. The symbols can be assigned to another memory location or to a basic scalar value. With the help of these symbols, static and tainted data is modeled and meta information, such as the data type, applied sanitization, encoding, or escaping is stored. The following data symbols are available:

- VALUE represents a static `"string"`, integer, float, or a resolved `CONSTANT`'s value. Defined constant values are stored in the environment.
- VARIABLE represents a `$variable` by its name.
- ARRAYFETCH represents the *access* of an array in the form `$x[y]` by its name x and the dimension y represented by data symbols. Multiple dimensions are possible, for example `$x[y][z]`.
- ARRAYKEY is used when the *key* of an array is explicitly accessed, e. g. in the loop `foreach($array as $key => $value)`. It is handled similarly to the VARIABLE symbol and is associated with the array's name. Those built-in functions, such as `array_keys()` and `array_search()`, that return the available keys in an array can be very precisely modeled with this data symbol.
- ARRAYTREE represents a newly declared array or the *assignment* of data to one array key (`$array[k] = $data`). It is organized in a tree structure. The array keys are represented by array edges which point to the assigned data symbol.
- VALUECONCAT represents the concatenation of two or more data symbols (`$a.$b`). Two consecutive VALUE symbols are merged to one VALUE symbol.
- MULTIPLE is a container for several data symbols. It is for instance used when a function returns different values depending on the control flow, or if PHP's *ternary* operator is used (`$c ? $a : $b`).

- Boolean is used in order to transfer the sanitization status of a symbol that is validated by a block edge. The details are explained in Section 3.5.
- Object is used when a new object is constructed by the keyword `new`. This data symbol is defined by the instantiated class' name and its properties. The properties are represented by a hash map that references a property name to a data symbol. By default, the map of properties in each Object symbol is empty.
- PropertyFetch models the *access* of a property. It extends the ArrayFetch symbol with a *property* dimension. This way, a PropertyFetch symbol is also capable of having an array dimension. The name associated to the symbol is the name of the receiving object. For example, the code `$v = $o->p[a]` assigns a PropertyFetch symbol with the name o, the property dimension p, and the array dimension a to the location v.

Each symbol (except for the basic symbol Value and the containers ArrayTree, Object, Concat, and Multiple) has a *type*, an *encoding*, and a *sanitization* status. By default, the type of each symbol is *string* and the symbol is not encoded nor sanitized. If a typecast is performed, we infer the new data type from the AST and assign it to the symbol. If the symbol's encoding is changed via built-in function, the encoding type is pushed to the symbol's encoding stack. On decoding, it is removed from this stack again. Moreover, each symbol can be sanitized against different types of vulnerabilities which are mapped to a vulnerability *tag*. These tags are assigned to symbols on sanitization. In Section 3.7.1, we discuss how the final sanitization status of a symbol is determined.

3.4.2 Block Summary

Data symbols assigned to a memory location are indexed in each basic block's *summary* by the location's name. The *block summary* stores the summarized data flow within one block. It is used to perform backwards-directed data flow analysis between multiple connected basic blocks throughout the CFG. By recursively looking up location names, data symbols can be resolved from previous basic blocks. Meta information, such as sanitization or encoding, is inherited from looked up symbols to resolved symbols. Our block summary is represented by the following properties:

- DataFlow maps assigned location names to the assigned *symbol*. In case of a defined array, the array name maps to an ArrayTree symbol whose keys can be fetched.
- ObjectCache references all recently and previously instantiated objects in form of Object data symbols. The ObjectCache is propagated forward from basic block to basic block. The details are explained in Section 3.4.4.
- PropwriteCache references all data assigments to object properties, where the receiving object is unknown at the time the assignment is analyzed. Then, the PropwriteCache is propagated between blocks, similarly to the ObjectCache. The details are explained in Section 3.4.5.

- Constants maps defined constant names to the assigned *symbol*. Uniquely defined constants with static values are stored in the environment for faster access during analysis.
- GlobalDefines records variable names which are put into global scope. These are later used to determine inter-procedural effects of a function (see Section 3.6.2).
- ReturnValue records the return value of the basic block. Note, that each basic block can have only one return symbol and the `return` statement is the last node in the nodes list. *Dead code* behind a `return` or `exit` statement is removed.
- RegisterGlobals states if the basic block enables register globals [40] due to built-in functions, such as `extract()` or `import_request_variables()`.
- IsExitBlock states if the basic block exits the program flow due to the `exit` or the `die()` operator, or by calling a user-defined isExitFunction (see Section 3.6.1).

3.4.3 Data Flow Analysis

In order to summarize the effect of a basic block in the *block summary*, the data flow in this block is analyzed. Based on a basic block's AST, we analyze all data assignments to memory locations of the form `loc := <assigned data>`. Other forms of data assignments are handled as well, but left aside for brevity reasons. The *assigned data* is transformed into *data symbols*. While we visit the nodes of each AST top-down, we keep track of the data type, encoding, and sanitization tags. Once the assigned data is transformed into data symbols, its memory location is indexed in the block summary for efficient lookups. In procedural PHP code, the assigned location `loc` is either a variable (e. g., `$x`) or an array dimension (e. g., `$x[y]`). Assigned data to a variable can be indexed in the block summary by the variable's name. Previously assigned data is overwritten. The assignment to an array dimension is handled by the wrapper symbol ArrayTree that stores assigned data in a tree graph. Dimension and data are both stored as data symbols. The tree structure allows efficient access to the data by providing one or multiple dimension(s) which are compared to the edges. The ArrayTree symbol is indexed in the block summary by the array's name. Further assignments to the same index extend its tree.

In order to summarize not only the data *assignment* but also the data *flow* of one basic block, the interaction between data assignments is evaluated based upon the current block's summary. For this purpose, the name of an assigned data symbol is looked up in the current summaries' index list to see if it can be *resolved* by previous definitions in the same basic block. A found Variable symbol is simply replaced with the symbol from the summary. An ArrayFetch symbol has to *carry* its array dimension to the resolved symbol. A resolved Variable symbol will turn into an ArrayFetch symbol with the carried dimension. The dimension of a resolved ArrayFetch symbol is *extended* by the carried dimension. In case the resolved symbol is an ArrayWrite, the symbol mapped to the carried dimension is fetched from the tree. The data flow analysis through objects and properties (*fields*) is more complex and is described in the next section. An example of two simple assignments is given in Listing 3.1.

```
1 $y = (int)$_GET['p'];
2 $z = $x . $y;
```

Listing 3.1: A basic block with two assignment nodes.

In the first assignment, an integer typecast is found that switches the data type of all subnodes to *int*. The subnode is evaluated to an ArrayFetch symbol. Finally, this symbol is mapped to the location `y` in the current block's DataFlow property. The ArrayFetch symbol has the name *_GET*, the type *int*, and one dimension with a Value symbol *p*.

In the second line, an assignment to location `z` is parsed. Here, a string concatenation is found and the left and the right part is evaluated. While the Variable symbol with the name *x* on the left remains unresolved for this block, the Variable *y* on the right can be resolved from the previously added ArrayFetch symbol in the DataFlow property `y`. Both symbols are added to a Concat symbol which is then mapped to the location `z` in the *DataFlow* property.

We do not model assignments by reference (*aliases*) in great detail yet since they are rarely used in modern PHP applications. However, we support function parameters *passed by reference* because these are sometimes used in custom sanitization functions and built-in functions, such as `array_walk()`. We handle these parameters in a similar way to global variables (see Section 3.6.1).

In case the location name is dynamic, backwards-directed string analysis is performed. If the result is one or more strings, then the assigned symbol is added to these location names. Otherwise, if the result stems from user input, a *Variable Tampering* warning is issued. A *variable* variable within the assigned expression is handled in a similar way.

3.4.4 Object-sensitive Analysis

When a new object is created, its constructor is analyzed. A constructor is either the `__construct()` method of the class or a method having the instantiated class' name. Our inter-procedural analysis ensures that all data assigned to properties within the constructor is assigned to the new Object symbol. The details are explained in Section 3.6.2.

Then, the created object is assigned to its memory location and indexed in the block summary as described in Section 3.4.3. During inter-procedural analysis of a method, the knowledge of all present objects and their corresponding class names is required. Thus, at the end of the simulation of one basic block all the indexed Object symbols are propagated to the next basic block into the ObjectCache (illustrated in Figure 3.2, *dotted* arrow). While our approach is aware of multiple different objects per code path, we assume for simplicity reasons that no cache index collides.

Moreover, we extract type information from type checks (e. g., `$o instanceof MyClass`) to determine missing class information. The class name is updated in the object cache or a *dummy* object is created if no related object is found.

The object cache is extended by each basic block when new objects are invoked and when all objects are propagated until the end of the CFG is reached. This way, each basic block has access to previously invoked objects within its CFG. If the CFG belongs to the

```
1 $text = 'test';
2 $obj = new MyClass;
3 if(...) {
4    $obj->data = $text;
5 }
6 echo $obj->data;
```

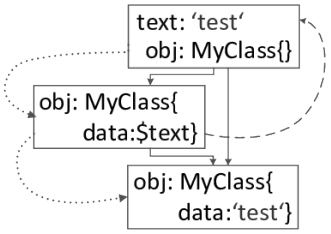

Figure 3.2: The code on the left creates a new object and assigns data to a property. The corresponding control flow graph is illustrated on the right. The created object `obj` is propagated forward throughout the CFG (*dotted* arrow). Data assigned to an object's property is resolved by backwards-directed data flow analysis (*dashed* arrow).

main code of a file, then the lifetime of all objects passes over. At this point, the object cache is emptied and the `__destruct()` method of each different instantiated class is being analyzed. The inter-procedural propagation of objects is explained in Section 3.6.2 and a case study is presented in Section 3.6.3 for better understanding.

Based on the object cache, special operations on locations pointing to objects are detected and the corresponding magic methods are analyzed (refer to Section 2.5.1). If the built-in functions `var_export()` or `serialize()` reference a memory location that points to an OBJECT symbol, then the corresponding magic methods `__set_state()` or `__sleep()` of the object's class are analyzed (if available). Similarly, the `clone` operator invokes the analysis of the method `__clone()` and an implicit or explicit typecast to *string* invokes analysis of the method `__toString()`. If an object is used within a dynamic function call, such as `$object()`, the method `__invoke()` of the object's class is analyzed.

3.4.5 Field-sensitive Analysis

Through knowledge about present objects, our approach can handle the access to properties. We model writes and reads to properties of objects (i. e., `$o->p`) in a similar way to the access of arrays. The challenge is to maintain object-sensitivity [116]. In the following, we refer to the accessed object `$o` as the *receiving* object, or in short, *receiver* [86].

Property Writes A property p of an object `$o` is written to if the location *loc* of the assignment *loc* := <*assigned data*> is a property access (i. e., `$o->p`). We then first try to resolve the *assigned data* by performing backwards-directed data flow analysis through all previously linked blocks' summaries (recall Section 3.4.3).

If the receiver's name `$o` is found in the object cache of the current basic block, then the assigned data's symbol is added to the property hash map of object `$o` in the object cache with the index p. In case an array dimension of a property is accessed (i. e., `$o->p[d]`), the assigned data is wrapped into an ARRAYWRITE symbol. An example is given in Figure 3.2. Here, the variable `$text` is resolved in line 4 and its value *test* is assigned to the object's property.

However, during intra-procedural analysis, the object cache is not always complete. For example, when object `$o` is a parameter or a global variable of the current function (see Listing 3.4), or when the receiver's name is the reserved variable `$this` that refers to the current object of the called method, then the receiver is unknown. In this case, we save the information about the receiver's name, the property dimension, and the assigned data symbol in the PropwriteCache of the current basic block. This cache is propagated through all upcoming basic blocks, similar to the ObjectCache. The details on how the property writes are assigned to the correct receiver during inter-procedural analysis are explained in Section 3.6.2.

Furthermore, we handle writes to *static* properties. Similar to the access of non-static properties, the receiver class can be related to the current callee's class (e. g., `self::$p` or `parent::$p`), or to a secondary class (e. g., `Class::$p`). In both cases, the target class name is determined from the class hierarchy and the assigned data is stored in the analysis' environment for later access.

Property Access A property p of an object `$o` is accessed if the location *assigned data* of the assignment *loc* := <*assigned data*> is an object's property (i. e., `$o->p`). The corresponding PropertyFetch symbol can be resolved from the block summary if the receiver name o is found in the object cache. First, the property dimension p is fetched from the hash map and then the array dimension a is carried to the resolved symbol. If the receiver name o is indexed in the data flow summary, then the receiver's symbol is fetched and the object's property dimension p is *carried* to it. In this process, a Variabe symbol is inferred into a PropertyFetch symbol with a property dimension p. An ArrayFetch symbol is inferred similarly, but it carries its array dimension to the PropertyFetch symbol. If a PropertyFetch symbol is resolved from the block summary into another symbol of this type, then the property dimensions are added. Finally, if the PropertyFetch symbol was not inferred from the block summary or the object cache, then it is looked up in the *propwrite* cache. Otherwise the PropertyFetch symbol remains unresolved.

Field-sensitive Magic Methods We also invoke analysis of magic methods for certain operations on PropertyFetch symbols. However, this is only possible when the class name of the receiver is resolved from the object cache. Then, if the built-in function `isset()` or `unset()` references to an inaccessible property (determined by the class definition), the magic method `__isset()` or `__unset()` of the receiver's class is analyzed. Furthermore, if the property dimension of a property read or write is not defined in the receiver's class, the magic method `__get()` or `__set()` is analyzed. When the receiver's class name cannot be resolved, no further analysis invoked. Note that in case of a POI vulnerability, an object of an arbitrary class is present so that field-sensitive magic methods are still supported for POP chain generation by considering all available classes (for details refer to Section 3.9).

3.4.6 Includes and Dynamic Code

As introduced in Section 2.1, *includes* are dynamic expressions in PHP and not static statements. *Includes* have a return value and can occur within conditions, assignments, or any other expression. In case the file name is not a static string, we try to reconstruct the

name of the included file. All entry edges of the current basic block are recursively visited and all possible values are constructed from the block summaries of previous blocks. If the reconstructed file name is ambiguous, then a regular expression is created and mapped to all available file names. If more than one file matches, we try to favor files in the same directory. Each possible included file is then handled as user-defined function that is called with empty arguments and with all local variables in global scope.

Eval operations are handled in a similar way. First, we try to reconstruct the evaluated string by backwards-directed string analysis using previous block summaries. If necessary, we decode multiple layers of encoded data (identified by the used built-in functions) to be able to also analyze obfuscated code. If we can parse the reconstructed string as PHP code, the code is handled in the same way as included PHP code. Dynamically generated code based upon unsanitized user input generates a *Code Execution* vulnerability report.

3.4.7 Built-in Data Flow Functions

As presented in Section 2.1, PHP is shipped with over 5 000 built-in functions. We simulate a total of 1 243 built-in functions, including 615 data flow functions, 77 validation functions (see Section 3.5), 360 sensitive sinks (see Section 3.7), and 191 uncategorized functions. To the best of our knowledge, this subset includes all essential functions for a comprehensive data flow and taint analysis. The data flow of 615 built-in functions is modeled through the usage of fingerprints. Each function is configured by name and its affected parameters. The data-flow functions can be categorized in the following seven groups:

- **alphanum (153):** Built-in functions, such as `strlen()` or `md5()`, effectively sanitize their argument by only returning alphanumerical values. These calls return a static VALUE symbol.

- **argument (116):** Other built-in functions, such as `trim()` or `strrev()`, return at least one of its arguments fully or partly. A flow of tainted data is possible through these functions and the symbols of these arguments are returned. For handling the conversion between arrays and strings, this category is divided into functions that return a string (68), an array (29), a single array element (10), or that split a string argument into an array (9).

- **escape (26):** Some built-in functions sanitize against certain vulnerability types by escaping meta characters. As introduced in Section 2.3.2, the function `addslashes()` sanitizes against SQL injection vulnerabilities with **s**ingle and **d**ouble **q**uotes by adding an escaping backlash. Thus, the sanitization tags `SQLI_SQ` and `SQLI_DQ` are assigned to the returned symbol, but intentionally not `SQLI_NQ` (**n**o **q**uotes).

- **substring (7):** String functions, such as `substr()` or `chunk_split()`, return a substring of an argument. This can destroy previously added escaping by cutting off an escaped meta character and then leaving behind an unescaped backslash. These functions are handled as *argument* functions but add a `SQLI_MI` tag. The tags help us identify SQL injections in queries with insufficiently escaped **m**ulitple **i**nput.

- **encode (18):** Other functions, such as `urlencode()` or `base64_encode()`, sanitize against all vulnerability types by encoding all meta characters. Thus, the encoding type is assigned to the argument symbol's encoding stack.

- **decode (25):** Built-in functions, such as `urldecode()` or `base64_decode()`, can turn harmless user data into malicious data. Thus, all previously added sanitization tags are removed from the returned symbol if the encoding stack is empty. If the decoding type matches the encoding type on top of the encoding stack, then the type is removed from the encoding stack.

- **callbacks (60):** Some built-in functions call other functions whose name is given as string argument. Examples are `array_walk()` or `set_error_handler()`. If the *callback* function's name can be reconstructed by string analysis, the function is analyzed intra- and inter-procedurally. If the function name is reconstructed only partly, then a regular expression is generated and performed on all available function names in order to identify a possible subset of functions which require analysis.

- **autoload (14):** Built-in functions, such as `class_exists()`, can invoke a user-defined *autoloader* function. If such a function is called with user input, then a vulnerability can be triggered in the autoloader function, for example an LFI or an RFI vulnerability.

- **database (83):** Various built-in functions, such as `mysql_fetch_array()` and `mysql_fetch_row()`, are used to detect the data flow through a database. The functions are separated in subgroups, depending if they fetch a table's row or column and if the returned data is an associative array or an object.

- **infoleak (113):** The built-in functions that return sensitive information about the web server's environment, such as `pq_version()` or `error_get_last()`, return a VALUE symbol that is flagged with an *infoleak* tag. If such a symbol is encountered during taint analysis of an XSS sink, such as `echo()`, an *information leakage* vulnerability is reported.

Another 191 frequently used built-in functions cannot be generalized in the above listed categories or need further processing. We model these functions by using abstract interpretation. For example, the built-in functions `htmlentities()` and `htmlspecialchars()` introduced in Section 2.3.2 sanitize input differently depending on their second argument. Thus, we first reconstruct the provided value in the second argument with string analysis and then add sanitization tags to the returned symbol accordingly. Other examples are built-in functions that use the format string syntax, such as `printf()` and `sprintf()`, that require in-depth analysis of the format string (see Section 2.1). Furthermore, certain built-in functions, such as `extract()` (see Section 2.1), can invoke data flow analysis as well as a taint analysis (see Section 3.7) simultaneously. Built-in functions not configured return the default value *1*.

3.5 Simulating Block Edges

Analog to the simulation of basic blocks, we simulate block edges when a basic block is connected to another basic block. In this simulation we try to identify generic and context-sensitive input validation (see Section 2.3) by analyzing the constraint of the jump, including 77 built-in functions. Each built-in function returns a BOOLEAN symbol that stores the validated data symbol, the required constraint *true* or *false* for successful validation, and the affected sanitization tags. With the help of these BOOLEAN symbols, input validation is recognized even when the validation status is part of the data flow, for example through a user-defined function's return value. The following types of input validation are recognized:

- **type checks (23):** A subset of built-in functions, for example `is_numeric()` or `ctype_digit()`, check if all the characters of its argument are numerical and return *true* on success. Thus, they can be used to avoid the presence of malicious characters inside their argument for a branch.
- **search (18):** Other built-in functions search within a given argument for specific strings and report on their finding. If the search string is a character in a configured set of characters required for exploitation of a specific vulnerabilities, then the sanitization tag of this vulnerability type is added to the data symbol.
- **file checks (13):** Some built-in functions, such as `is_file()` or `stat()`, generically validate a given argument by checking if it is a valid file on the file system. If no functionality of uploading a file with an arbitrary file name is available, then these functions effectively validate their argument against all but file vulnerabilities.
- **regex (8):** A symbol can be checked for a range of characters with the help of regular expression patterns. These checks are performed with built-in functions, such as `preg_match()` or `ereg()`. We transform the regular expression into an AST and check for every *or* branch if a configured set of characters can pass the expression. Each character is associated with different sanitization tags that are added to the target symbol of the regex function if the character cannot pass the regular expression. While our approach is not sound for all regular expressions, our evaluation has shown that most of the regular expressions used for sanitization are kept simple and no false positives or negatives were encountered.
- **compare (7):** A symbol can be validated before entering a basic block if it is compared to a static value in the entry edge. Built-in functions, such as `strcmp()`, compare the first argument against the second argument and return `0` if both arguments match. If one argument is a static value, the other argument is effectively limited to this value and therefore validated.
- **operators (6):** Next to built-in functions, a symbol can be compared to a static value by an comparison operator, such as the `Equal` (`==`) or `Identical` (`===`) operator. The nodes `NotEqual` (`!=`) and `NotIdentical` (`!==`) only validate their arguments if they are subnodes of a `BooleanNot` node. Additionally, a symbol is validated within a basic block if the entry edge requires the symbol to be `empty` or not set (`!isset`).

- **length (5):** Another form of input validation occurs when the length of data is limited to an amount of characters that is not sufficient for exploitation. This check can be performed with built-in functions, such as `strlen()`. If the return value is compared to a number lower than `3`, then we assume a sufficient input validation.
- **whitelists (3):** The built-in functions `array_search()`, `array_key_exists()`, and `in_array()` are often used to check whether a value is in a given set of allowed values or not. The tested symbol is then added as validated symbol to the block edge. Another form of whitelisting is recognized when a specific array key is checked for presence (`isset($whitelist[$check])`).

Moreover, user-defined functions can *wrap* the validation mechanisms listed above. For example, the function `isValid()` in Listing 3.3 returns *true* if the argument is numerical, and *false* otherwise. In order to detect correct validation and to prevent false positives, the data flow through the function `isValid()` is analyzed. If the return symbol of a basic block is a constant with the value *true* or *false* (line 3), we check if the entry edge of the basic block validates a symbol which is at the same time a parameter of the analyzed function. Then, a BOOLEAN symbol is connected to the validated parameter and to the boolean value and is added to the RETURNVALUES of the function (see Section 3.6.1).

```
1 function isValid($value) {
2    if(is_numeric($value)) {
3       return true;
4    }
5    return is_numeric($value); // false
6 }
```

Listing 3.3: Sanitization with a user-defined function.

3.6 Procedural Analysis

While the data flow of built-in functions is known and configured, the data flow and side effects of user-defined functions and methods require an analysis of its code. This is performed during *intra-procedural analysis*, introduced in Section 3.6.1. The analysis results are summarized in a *function summary*. As described in Section 3.6.2, this summary is then reused during *inter-procedural analysis* at the call-site of every function call [115]. The inter-procedural analysis of methods is a challenging task because the receiver and the corresponding class name of the method need to be determined first.

3.6.1 Intra-procedural Analysis

If a user-defined function is called for the first time during an analysis, then a new CFG of the function is created. As described in the last section, the CFG will consist of simulated basic blocks with block summaries. A *function summary* is created that summarizes the data flow of *all* basic blocks in the CFG and is then saved to the environment. For this purpose, a depth-first search through all basic blocks of the CFG is initiated. If a basic block has no outgoing edges, it either has a `return` statement, an `exit` statement, or it

is the last block in the CFG. Based on these *end blocks*, a function summary with the following properties is created:

- ReturnValues: If the *end block* has a returnValue, its symbol is traced through previous basic blocks and all resolved symbols are added as function return value. This may include function parameters.
- ChangedGlobalVars: For each *end block* in the CFG, the possible set of altered global variables is generated by tracing globalized variables backwards to the basic block that put it into global scope. These variables are stated in the GlobalDefines summary, as introduced in Section 3.4.2.
- IsExitFunction: Indicates if the function exits the program flow. This is the case if *all end blocks* are flagged as isExitBlock.

During taint analysis within a user-defined function (see Section 3.7), the following properties can be added to the function summary. Their values are mapped to the vulnerability type of the current taint analysis.

- SensitiveParams lists the function's parameters that flow into a sensitive sink.
- SensitiveGlobals lists the local variables that are fetched from global scope and flow into a sensitive sink.
- SensitiveProperties lists the properties that are fetched from an object and flow into a sensitive sink.

Recursive function calls are not handled for now. While this introduces unsoundness, we are not aware of a real-world web security vulnerability that only occurs within a certain level of recursion. Furthermore, since our algorithm is path-insensitive, such a vulnerability would most likely be detected.

3.6.2 Inter-procedural Analysis

After a user-defined function or method was simulated, the inter-procedural effects of the call can be evaluated and changes to the current scope can be processed. For example, if the function or method was marked as an IsExitFunction during simulation, the basic block of this call is consequently exiting the program flow. Thus, all outgoing edges and the upcoming dead code are removed from the basic block. Furthermore, the *ObjectCache* and *PropwriteCache* is propagated from the function summary to the callee's basic block. However, objects are only propagated if their receiver is a global variable or a return value. Other objects are deleted from the cache and their destructor is invoked. Property writes are applied to global receivers as well as to receivers that were passed by parameter. The receiver name is adjusted to the arguments of the call. Property writes of a method to the receiver `$this` are applied to the receiver of the method call. Also, all changed global variables are copied to the basic blocks DataFlow property. Then, all SensitiveParams, SensitiveGlobals, and SensitiveProperties are adjusted to the callee's arguments and traced backwards, starting from the call-site's basic block. The details of the taint analysis are explained in Section 3.7.

Challenge: Receiver Analysis Because a method name can be defined in multiple classes, we have to determine the receiver's class in order to invoke the analysis of the correct method [116]. A call to a static method is easily mapped to the correct class by its specified name (e. g., `Class::method()`). In case the static keywords `self::method()` or `parent::method()` are used, then the class name can be resolved from the class hierarchy of the current method's class [34, 127]. The same applies if the reserved variable `$this` is used as receiver.

For all other non-static method calls, such as `$o->method()`, the class name has to be inferred from the receiver variable `$o`. If the receiver's name is found in the current block's object cache, then the class name is extracted from the cached OBJECT symbol. Note that the object cache contains only objects that were created in the current CFG or imported into the current CFG as return value of a function. However, as shown in Listing 3.4, if the receiver is passed as an argument (`$obj1`) or as a global variable (`$obj2`) to the currently analyzed method, no information about the receiver is available. The callee's context is only applied to the function summary, while our intra-procedural analysis is context-insensitive.

```
1 public function handler($obj1) {
2    $obj1->method1(1, 2);
3    global $obj2;
4    $obj2->method2(1, 2, 3);
5 }
```

Listing 3.4: Receiver `$obj1` and `$obj2` are unknown.

We approach the problem for `$obj1` by searching for all the available methods named `method1()` in all class definitions. If the name is unique, the corresponding method is invoked. Otherwise, we compare the number of arguments (here: two) to the number of parameters specified in the method declarations. Then, we invoke the analysis for all matching candidates and combine their function summaries to one summary. While this approach can potentially lead to an over-approximation, it is likely that methods, such as the method `handler()` in Listing 3.4, are intended to call different methods on different objects.

For `$obj2` we take a different approach. In our initial setup phase, we index the name of all global variables within all application's functions and methods identified by the `global` keyword or `$GLOBALS` variable. If a new object is assigned to a location having one of these indexed names, the object's class name is referenced to the index. During intra-procedural analysis, the class name can be then retrieved for global variables. In case of *dynamic* global variables we fall back to the approach as described for `$obj1`.

For static method calls we check the accessibility of the method regarding to the receiver's class name with the help of our class hierarchy. We invoke the analysis of any defined `__staticCall()` method of that class if the called method is not accessible. The same applies to the `__call()` method for non-static method calls. Similar to the analysis of field-sensitive magic methods, our approach is limited by the success of our receiver analysis. However, during object injection, all classes are considered so that our analysis of invocation-sensitive magic methods for gadget chain generation is not limited.

3.6.3 Case Study: OOP Code Analysis

We now discuss the analysis of a real-world OOP code in order to illustrate our novel approach. Our analysis begins in line 12 of Listing 3.5, where a new OBJECT symbol is created and indexed in the block summary under the name *objPagePicker*. We neglect the constructor analysis. In the next line, the method `run()` is called. Its class is determined from the recently indexed OBJECT symbol.

```
1  class PagePicker extends Backend {
2     public function run() {
3        if ($_POST && Environment::get('isAjaxRequest')) {
4           $this->objAjax = new Ajax(Input::post('action'));
5        }
6        ...
7        if ($_POST && Environment::get('isAjaxRequest')) {
8           $this->objAjax->execPostActions($objDca);
9        }
10    }
11 }
12 $objPagePicker = new PagePicker();
13 $objPagePicker->run();
```

Listing 3.5: The method `run()` of the class `PagePicker`.

Our analysis continues intra-procedurally in the first basic block of the method `run()` in line 4. Here, a new object of the class `Ajax` is instantiated and assigned to the property `$this->objAjax`. Again, we omit the constructor analysis. The receiver `$this` is unknown at that time. Thus, we store the new object into a PROPWRITE symbol. It assigns the OBJECT symbol `Ajax` to the property `objAjax` of the receiver `this`. The PROPWRITE symbol is stored in the PROPWRITE cache and propagated to each further basic block within the method `run()`. Consequently in line 8, the receiver `$this->objAjax` of the call `execPostActions()` is resolved to the `Ajax` object from the PROPWRITE cache. After this call, the analysis of `run()` terminates and the property write to `objAjax` is applied to the receiver `$objPagePicker`.

```
1  class Ajax extends Backend {
2     public function execPostActions(DataContainer $dc) {
3        if ($dc instanceof DC_Table) {
4           echo $dc->editAll($this->ajaxId, $id);
5        }
6     }
7  }
```

Listing 3.6: The method `execPostActions()` of the class `Ajax`.

In Listing 3.6, the executed method `execPostActions()` is shown. Due to our context-insensitive intra-procedural analysis, arguments which are passed to a method are unknown during analysis time. Thus, the receiver `$dc` of the call `editAll()` in line 4 is unknown. However, we are able to infer the class information from the parameter specification (*DataContainer*) and more specifically from the `if`-constraint in line 3 (*DC_Table*). Otherwise, the correct method would have been found by *method fingerprinting*. There are two methods defined with the name `editAll()`, but only one accepts two parameters by its specification.

3.7 Taint Analysis

When simulating a basic block, each function call is inspected for potential vulnerabilities. We identified 360 sensitive sinks in the PHP language which we configured with function name, sensitive parameter, and vulnerability type. For each called sensitive sink, a new backward-directed taint analysis [109] is invoked for the corresponding vulnerability type that tries to determine if a source is used within the sink. Compared to an exhaustive forward-directed analysis of all sources, our demand-driven taint analysis of sinks potentially explores fewer program paths. Our approach supports 36 different vulnerability types that are listed in the following. These are refined to 45 different markup contexts, e. g., a *file inclusion* can be refined to a *local* or *remote file inclusion*.

1. Autoload Injection
2. Code Execution
3. Command Execution
4. Cross-Site Scripting
5. Denial of Service
6. Directory Listing
7. Env. Manipulation
8. Exec. After Redirect
9. File Create
10. File Delete
11. File Inclusion
12. File Upload
13. File System Manip.
14. File Write
15. HTTP Response Split.
16. Information Leakage
17. Library Injection
18. LDAP Injection
19. Log Forgery
20. Mass Assignment
21. Memcached Injection
22. NoSQL Injection
23. Open Redirect
24. Path Traversal
25. PHP Object Injection
26. PHP Object Instant.
27. Reflection Injection
28. Resource Injection
29. Server-side JS Injection
30. S.-side Request Forgery
31. Session Fixation
32. SQL Injection
33. Variable Manipulation
34. Weak Cryptography
35. XML/XXE Injection
36. XPath/XQuery Inj.

First, all possible strings that *flow* into the sensitive argument are reconstructed through backwards-directed data flow analysis. Furthermore, each string is inspected in a context-sensitive way for user input. If unsanitized user input was found and the markup context is exploitable, then a new vulnerability is reported.

3.7.1 Data Flow Analysis

In order to find all possible values of a sensitive sink's argument, the argument (from now on referred to as *traceSymbol*) is traced backwards through all basic blocks which are referenced as an entry to the current basic block. In our implementation, we loop through all entry edges of the current basic block that do not sanitize the *traceSymbol* and look-up its name in the DataFlow property of each block summary. If a match is found, the *traceSymbol* is replaced with the mapped symbol and all sanitization tags and encoding types are copied. Then, the trace continues through all linked entry edges of the basic block. Finally, the unique composition of all return values in the CFG are returned.

The algorithm stops if the *traceSymbol* maps to a static `Value` symbol or if the current basic block has no entry edges. If the *traceSymbol* is a variable or an array access, it is checked if the *traceSymbol* is in the list of the 13 superglobal variables (see Section 3.7.3).

It is also ensured that the *traceSymbol* is of type *string*, that it is not encoded, and that it is not sanitized against the currently analyzed vulnerability type.

The sanitization status of a symbol is inferred as follows: if a symbol is encoded, for example with the *base64*, *hexadecimal*, or *zlib* encoding, then it is sanitized against all vulnerability types. In case a symbol is decoded and was previously encoded with the same encoding, its sanitization status depends on previously added sanitization tags. If a symbol is decoded without prior encoding, then all sanitization tags are dropped because malicious characters can be provided in an encoded way by an attacker.

If the *traceSymbol* maps to an unsanitized tainted source, the *traceSymbol* is saved and a linked user input tag is returned. Otherwise, if the *traceSymbol* maps to an unsanitized parameter or globaled variable of the user-defined function that the basic block is part of, a corresponding tag is returned. These tags can be analyzed context-sensitively later on.

In order to optimize this time-intense process, we implemented caching of the result for each basic block. When a symbol is traced through a basic block, the result is cached within the basic block. If the same symbol is traced through this basic block again, then the result is retrieved from the cache and the trace is aborted. This drastically improves the performance of our analysis. Moreover, we configure a *maximum* amount of traversed edges that introduces a path limit in order to optimize performance. Although this can lead to false negatives, we did not encounter these in practice. Moreover, step-by-step caching of lookup results for each basic block raises the chance that a full path analysis is not required if parts of the analysis were previously analyzed.

3.7.2 Context-Sensitive Markup Analysis

The obtained strings from the data flow analysis are analyzed for user input tags. For each vulnerability type, a different analyzer is invoked that identifies the context within the markup. Depending on the context, specific vulnerability tags are determined. Only if the *taint symbols* are not sanitized against the current vulnerability tag, they are marked as a tainted symbol and a vulnerability is issued.

If no user input was found, but the analyzed sensitive sink is called within a user-defined function, the strings are analyzed for parameter and global tags. When these are found in one of the strings, the corresponding symbols are added as vulnerable parameters or as vulnerable global variables to the user-defined function summary. During inter-procedural analysis these symbols are analyzed starting from the basic block of the function call. In the following, we explain the analysis of two markups. Further *markups*, such as HTTP headers or file names, require unique but less complex analysis.

HTML For each XSS vulnerability, we inspect the HTML markup of the reconstructed string. The HTML markup is resolved from previous basic blocks similar to the techniques described by Minamide [87] and used by Wasserman and Su [149]. Each reconstructed string is parsed with an HTML parser into a structured HTML DOM tree. First, the text between two HTML elements is searched for user input tags. On success, the vulnerability tag is changed to `XSS_ELEMENT`. If the HTML tag name is *script* or *style*, the vulnerability tag is changed to `XSS_SCRIPT` or `XSS_STYLE` accordingly. While sanitization of meta

characters within a normal HTML element is sufficient, it does not prevent attacks when these characters are injected into a `script` or `style` tag.

If the HTML element has attributes, each attribute's value is searched for user input tags. Depending on how the attribute value is quoted, the vulnerability tag `XSS_ATTR_DQ`, `XSS_ATTR_SQ`, or `XSS_ATTR_NQ` is set for a **d**ouble, **s**ingle, or **n**ot-**q**uoted value. As discussed in Section 2.3.2, it is also important to consider the type of the HTML attribute. A list of 49 eventhandler and 21 url attributes is configured to set the sanitization tag `XSS_ATTR_JS` or `XSS_ATTR_URL`.

SQL As described in Section 2.3.2, a SQLi vulnerability is also context-sensitive. Our SQL parser tries to determine if the injection happens between a **s**ingle **q**uoted value (`SQLI_SQ`), **d**ouble **q**uoted value (`SQLI_DQ`), or is embedded into the SQL query unquoted (`SQLI_NQ`). The sanitization tag `SQLI_MI` (**m**ultiple **i**nput) is reserved for symbols that are sanitized by escaping quotes but were passed through a *substring* built-in function afterwards. This can lead to a SQL injection vulnerability if the substring reveals a trailing backslash and more than one tainted source flows into the sensitive sink.

3.7.3 Source Analysis

For further improvement, we analyze the tainted source depending on the vulnerability type. For example, client-side vulnerabilities such as *session fixation* or *HTTP response splitting* require an easy to forge source for practical attacks. Thus, tainted values originating from uploaded file names (`$_FILES`), cookies (`$_COOKIE`), or HTTP headers (`$_SERVER['HTTP_*']`) are ignored. While all HTTP headers stored in the superglobal `$_SERVER` array can be altered by the user arbitrarily, there are several CGI parameters that disallow certain characters and are not practical for exploitation of certain vulnerability types. Examples for the `$_SERVER` key's limitations are:

- `HTTP_HOST`: A slash or a backslash within the *Host* header is disallowed and will result in a bad request blocked by the web server. Thus, the *Host* header cannot be used for *Path Traversal* attacks.
- `PHP_SELF`, `PATH_INFO`: A *path traversal* attack within the requested path will result in a *path traversal* attack against the web server and will most likely fail.
- `PHP_SELF`, `PATH_INFO`, `REQUEST_URI`: The requested path and URI contains the current path as prefix. Consequently, these keys cannot be used to inject protocol handlers to an URL attribute or to exploit a *remote file inclusion* vulnerability because both attacks require the control of the first injected characters.

Note, that the source `$_SERVER['QUERY_STRING']` and `$_SERVER['REQUEST_URI']` are not listed with further limitations. Although browsers such as *FireFox* and *Chrome* automatically urlencode meta characters within the query string, *Internet Explorer* does not. Furthermore, these sources can be arbitrarily tainted by manually crafting an HTTP request. Our approach is also aware that `$_GET`, `$_POST`, and `$_COOKIE` parameters can be supplied as arrays by the user. Hence, we not only mark all parameter values as tainted, but also all available key names.

3.7.4 Environment-aware Analysis

A PHP application and its vulnerabilities may behave differently depending on the PHP configuration. For this purpose, we configure four different PHP settings and a PHP version number. The version number is important to categorize certain file-based vulnerabilities that base on null-byte injections or *HTTP response splitting* attacks that were fixed by the PHP developers. Furthermore, the PHP settings *magic_quotes_gpc*, *allow_url_fopen*, and *allow_url_include* may restrict certain vulnerabilities, while the PHP setting *register_globals* may introduce certain vulnerabilities. We also aim to detect reimplementations of these settings, for example, when sanitization is applied to the superglobals or the built-in function `extract()` is used.

3.8 Second-order Taint Analysis

Taint analysis and similar code analysis techniques are used to study the data flow of untrusted (also called *tainted*) data into critical operations of the application. However, web applications can also store untrusted data to external resources and later on access and reuse it, a problem that is overlooked in existing approaches. The main problem is to decide whether data fetched from these resources is tainted or not. Assuming all data to be tainted would lead to a high number of false positives, while a conservative analysis might miss vulnerabilities. Since the data flow is deferred and can be split among different files and functions of the application, second-order vulnerabilities are difficult to detect when analyzing the source code statically. Furthermore, static code analysis has no access to the real data that is stored in external resources and used by the application.

In this section, we introduce a refined type of taint analysis. During our data flow analysis, we collect all locations in persistent stores that are written to and can be controlled (tainted) by an adversary. If data is read from a persistent data store, the decision if the data is tainted or not is delayed to the end of the analysis. Eventually, when all taintable writings to persistent stores are known, the delayed decisions are made to detect second-order vulnerabilities. The intricacies of identifying the exact location within the persistent store the data is written to is approached with string analysis. Furthermore, sanitization through database lookups or checks for existing file names are recognized.

3.8.1 Overview

We now introduce our novel approach to detect second-order vulnerabilities. The data flow is illustrated in Figure 3.3 (b). Contrarily to a conventional taint-style vulnerability as shown in Figure 3.3 (a), a source flows into a PDS before it flows from the PDS into a sensitive sink. We model the data that is read from a PDS by new data symbols δ^* that hold information about their origin.

During code analysis, *taintable* PDS are identified. They are stored together with the minimum set of applied sanitization and encoding tags of the tainting data symbol δ. If a data symbol δ^* is encountered unsanitized during the taint analysis of a sensitive sink, a vulnerability report is created if its originating PDS was identified as taintable.

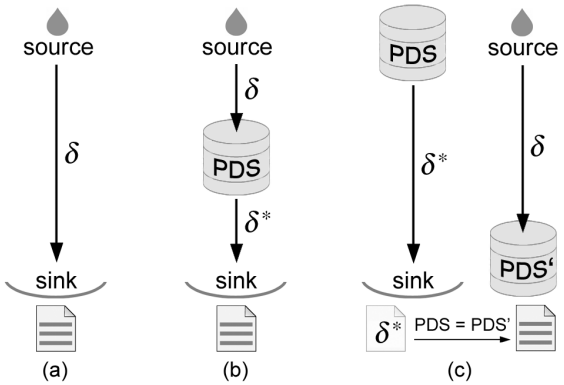

Figure 3.3: Data flow model of a conventional (a) and a second-order (b, c) vulnerability.

If the PDS is not known as taintable, a *temporary* vulnerability report is created, as shown in Figure 3.3 (c). The report is connected to the data symbol δ^*. At the end of the code analysis, we decide if the data symbol originates from a taintable PDS by comparing its origin to all collected taintable PDS.

In the following, we introduce the analysis of writings to different PDS. Furthermore, our new data symbols δ^* are introduced that model the reading and access of data that is stored in PDS.

3.8.2 Databases

Modeling the data flow through databases is a complex task, mainly due to the large API that is available for databases and the usage of a query language. First, we try to obtain as much knowledge of the SQL schema as possible. Then we try to reconstruct all SQL queries during SQL injection analysis of 110 built-in query functions. Finally, the type of operation is determined, as well as the targeted table and column names. The access of data is modeled by new data symbols.

Preparation During the initialization of our analysis, we collect all files with a *.sql* extension. All available `CREATE TABLE` instructions within these files are parsed so that we can reconstruct the database schema, including all table and column names as well as column types and length. If no schema file is found, each PHP file in the project is searched via regular expression. The knowledge of the database schema improves precision when data is read in an *unspecified* way, or when data is sanitized by the column type or length.

Writing A write operation to a database is detected if the SQL parser identifies an `INSERT`, `UPDATE`, or `REPLACE` statement. By tokenizing the SQL query, we determine the targeted table's name, all specified column names, and their corresponding input values. In case of an *unspecified write*, the parser makes use of the database schema. If an input value of a column contains tainted data (see Section 3.7), the affected column and table name is marked as *taintable* together with the linked source symbol and its sanitization tags.

Reading If the SQL parser encounters a `SELECT` statement, we try to determine all selected column and table names. Multiple table names can occur if tables are *joined* or *unioned*. Alias names within the query are mapped and resolved. In case of uncertainty, the parser makes use of the database schema. Finally, a new `ResourceDB` symbol is mapped to the analyzed query function as return value. This symbol holds information about all selected column names in a numerical hash map and its corresponding table names.

Access In PHP, database result resources are transformed into arrays by built-in fetch functions (refer to Listing 2.36). We ignore the mode of access and let 89 configured fetch functions return a `Variable` symbol with the name of the resource. When an `ArrayDimFetch` symbol accesses the result of these fetch functions, it is inferred to the corresponding `ResourceDB` symbol. In this case, the carried dimension of the `ArrayDimFetch` symbol is evaluated against the available column names in the `ResourceDB` symbol. If the *asterisk* character is contained in the column list and the dimension is numerical, the database schema is used to find the correct column name. Otherwise, if the dimension equals a column name in the field list, a new `DataDB` symbol is returned that states which column of which table is accessed.

Sanitization Certain implicit sanitization is considered when dealing with SQL. If a column is compared to a static value within a `WHERE` clause in a `SELECT` statement, the return value for this column is sanitized. In this case, the static value is saved within the `ResourceDB` symbol and mapped to the column as return value. Furthermore, a sanitization tag for the used quote type is removed when data is updated or inserted to the database because one level of escaping is lost during writing.

3.8.3 Session Keys

The analysis of session variables does not require a complex markup parser or new data symbol. Instead, session data is handled similar to other global arrays. Taintable session keys are stored during the analysis phase.

Writing If data is assigned to a `Variable` or `ArrayDimFetch` symbol during block simulation and the symbol's name is `$_SESSION`, the assigned data is analyzed via taint analysis. If the assigned data is tainted, its resolved source symbol is stored into an `ArrayDimTree` symbol in the environment, together with the dimension of the `$_SESSION` symbol. This way, an `ArrayDimTree` is built with all taintable dimensions of the session array that link to the tainted source symbols and their corresponding sanitization tags.

Reading The access to session data is modeled by `ArrayDimFetch` symbols with the name `$_SESSION` and requires no modification. During taint analysis inside a user-defined function, session variables are handled as global variables. They are added to the function summary and they are inspected for each function call in a context-sensitive way. This avoids premature decisions about the taint status inside a function if the session key is overwritten before the function is called. Just as for a `DataDB` symbol, a *temporary* vulnerability report is created if a `$_SESSION` variable taints a sensitive sink.

3.8.4 File Names

To detect taintable file names, we collect file paths a user can write to. For this purpose, new data symbols model directory resources and their accesses. Whenever a path is reconstructed only partially, we use the same approach as in file inclusion analysis. Here, a regular expression is created and mapped to all available paths that were detected when loading the application files.

Writing To detect a file name manipulation with user input, we analyze 27 built-in functions such as `copy()`, `rename()`, and `file_put_contents()`. Additionally, file uploads with `move_uploaded_file()` are analyzed. Note that at the same time these built-in functions are sensitive sinks and generate vulnerability reports such as an *arbitrary file upload* vulnerability. The path argument is analyzed by conventional context-sensitive string analysis. If the path is tainted, we store it with its prefix as taintable. When no prefix is present, the file path of the currently analyzed file is taken. Additionally, if the source is not sanitized against *path traversal* attacks, all paths are assumed as taintable and a flag is set during analysis accordingly.

Reading We handle three different ways of opening a directory with PHP's built-in functions. First, we model the built-in function `scandir()` that returns an array, listing all files and directories within a specified path. Second, we model the built-in function `glob()` that also returns an array that lists all files and directories specified by a pattern. We transform the pattern into a regular expression by substituting the pattern characters `*` and `?` into regular expression equivalents. Third, we model the built-in function `opendir()` which returns a *directory handle*. For all mentioned built-in functions, we reconstruct the opened path by string analysis and return a `ResourceDir` symbol that stores the path's name.

Access The returned result of `scandir()` and `glob()` is accessed by an array key. Since we do not know neither the amount nor the order of files in a directory, we return a `DataPath` symbol whenever a `ResourceDir` symbol is inferred from an `ArrayDimFetch` symbol, regardless of its dimension. For this purpose, we let the built-in function `readdir()` that is supposed to read an entry of a *directory handle* return an `ArrayDimFetch` symbol with an arbitrary dimension and the name of the directory handle. It is inferred to a `DataPath` symbol when the trace of the `ArrayDimFetch` symbol results in a `ResourceDir` symbol.

Sanitization In order to model sanitization that checks if a given string is a valid file name, 11 built-in functions such as `file_exists` and `is_file()` are simulated. We modified the sanitization check in a way that these functions only sanitize if there is no taintable file path found. For this purpose, a flag is set during taint analysis if sanitization of a source by file name is detected. The flag issues only a temporary vulnerability report that is revised at the end of the analysis regarding the ability to taint a file path.

3.8.5 Multi-Step Exploits

In order to detect multi-step exploits, we store all table names of all writing SQL queries that are affected by SQLi. Furthermore, we set a flag during the analysis process if an *arbitrary file write* or *arbitrary file rename* vulnerability is detected. At the end of the analysis, when the taint decision is made for data that comes from a PDS, multi-step exploit reports are added to the initial vulnerability. This is done for all vulnerabilities that rely on a `DataDB` symbol that is not tainted through second-order but which table name is affected by SQLi. Also, a multi-step exploit is reported if a `DataDir` symbol occurs and the flag for a *file rename* vulnerability was set. All session data is treated as tainted if an *arbitrary file write* vulnerability was detected. Additionally, any *local file inclusion* vulnerability is extended to a *remote code execution* if a file write or upload feature is detected. Moreover, a SQLi vulnerability within a `SELECT` query returns a `DataDB` symbol with a *taint* flag. This flag indicates that all accessed columns are taintable by modifying the `SELECT` query during an attack. Thus, all columns of the `DataDB` symbol are taintable.

3.9 POP Chain Generation

By performing static code analysis that supports the analysis of PHP's OOP features, we are able to collect sensitive sinks in the application's code that can be reached after a PHP object was injected. More specifically, we can leverage our inter-procedural, field-sensitive, and object-sensitive data flow analysis to analyze the relevant OOP features and to construct an actual attack payload for each detected gadget chain. The resulting chains allow us to verify the ability to exploit a potential POI vulnerability.

3.9.1 Approach

Whenever our analysis reports a call to `unserialize()` as vulnerable, the return value of the `unserialize()` call is an Object symbol with a special *POI* flag set to *true*. If the return value of this `unserialize()` call is assigned to a variable, the flagged *Object* symbol is added to the current block's object cache that is propagated through the upcoming basic blocks, as described in the previous section. However, its flag causes certain different analysis steps regarding calls to magic methods.

First, all `__wakeup()` methods of all classes are analyzed as *initial gadgets*. If an object-sensitive magic method is invoked on a flagged Object symbol, all magic methods of its type are also analyzed. This applies as well to a field-sensitive or invocation-sensitive magic method that is invoked on a flagged Object symbol as receiver. The inter-procedural analysis of the magic methods is performed with an important difference: All sensitive properties of the function summary immediately report a POP gadget chain because the attacker has control over the object's properties.

Furthermore, we limited gadget chains to only severe vulnerabilities by deactivating the detection of client-side vulnerabilities, such as cross-site scripting and open redirects, in our approach. We also omit vulnerabilities that are triggered by a context-independent magic method and cannot be exploited, such as path traversal attacks against file han-

dlers without further processing. An exemplary POP analysis and report is presented in Section 3.9.3.

Our approach is also aware of a limitation to certain magic methods if the deserialized object is checked with `instanceof ClassName` throughout the code path. As described in Section 3.4.4, the type information of the deserialized object is then updated. Thus, only magic methods of the class `ClassName` are invoked and analyzed.

3.9.2 Challenges

Our approach has two remaining challenges. Recall Listing 3.4 where an object is unknown at intra-procedural analysis time. If we assume that `method1()` or `method2()` is a magic method, we do not know at the time of the intra-procedural analysis if the object is flagged or not. Thus, we do not know if all magic methods should be analyzed or not. We approach this problem by setting a different flag for each invoked magic method on an unknown receiver in the function summary. When a method is called with a flagged object as argument, we can tell from the function summary during inter-procedural analysis which magic method was invoked and we trigger its analysis.

A false gadget chain report occurs if a magic method of a class that is shipped with the project is analyzed, although the class is not loaded at runtime within the executed code path. We approach this problem by creating a stack of included files [55] during analysis *on-the-fly*. Before a magic method is analyzed, the file name of the method's class is confirmed in the stack in order to prove its availability. This routine is ignored if a class autoloader is detected [132].

3.9.3 Case Study

We now introduce a previously unreported gadget chain in Contao CMS leading to an arbitrary file delete vulnerability. The chain is invoked through the `__destruct()` method of the class `Swift_Mime_SimpleEntity` that is available through an autoloader. This initial gadget is shown in Listing 3.7 and it is automatically analyzed, when the flagged Object symbol of a POI is removed from the object cache. In line 3, we invoke the analysis of all available `clearAll()` methods within the application's code base because the receiver `$this->_cache` is unknown. It can be arbitrarily specified during object injection and point to any `clearAll()` method.

```
1 class Swift_Mime_SimpleEntity {
2    public function __destruct() {
3       $this->_cache->clearAll();
4    }
```

Listing 3.7: Initial POP gadget in Contao CMS.

There are four `clearAll()` methods available in the code base. While three of them are harmless, the one in the class `Swift_KeyCache_Disk` triggers another gadget. As shown in Listing 3.8, in line 3, it calls the function `clearKey()`. The receiver of this call is the reserved variable `$this`. Thus, only methods within the same class or its class hierarchy are considered and the method defined in line 5 is the only candidate.

```
1 class Swift_KeyCache_Disk {
2    public function clearAll()
3       $this->clearKey();
4    }
5    public function clearKey()
6       unlink($this->_path);
7    }
8 }
```

Listing 3.8: Final POP gadget leading to arbitrary file delete.

Here, the property `_path` is used in the sensitive built-in function `unlink()` that deletes a file. We transfer the sensitive property `_path` to the receiver `$this->_cache` in the method `__destruct()`, where it issues a vulnerability report as shown in Listing 3.9. The POP chain report is then attached to the POI vulnerability report.

```
Unserialize() to File Delete (unlink)
 Swift_Mime_SimpleEntity::__destruct()
 Swift_Mime_SimpleEntity->_cache = Swift_KeyCache_Disk
 unlink(Swift_KeyCache_Disk->_path)
```

Listing 3.9: Example of a generated POP chain report.

3.10 Limitations

Our approach is affected by the generic limits of static code analysis, as well as limitations that stem from our design. As in all different approaches, a solid tradeoff between true positives, false positives, and performance is required.

A general drawback of SCA is that it works with static compile-time data. The runtime data types and values are uncertain and their flow is simulated based on assumptions. Likewise, dynamic data that depends on the program's input, environment, or external data, such as from databases or files, is unknown during static analysis. With the growth of code complexity and inaccurate assumptions, the simulation is losing precision and false positives occur [148]. More specifically, our analysis can mistake in handling reflections configured in external files, in template engines that combine data with file content, or in complex string constructions within loops as used by SQL query builders. These limitations can potentially lead to false negatives and false positives, particularly during the analysis of frameworks. From a broader perspective, our approach can be broken down to the problem of statically reconstructing all strings that can be generated at runtime by an application and thus is limited by the halting problem [142].

Furthermore, static analysis is infeasible to examine all program paths due to the problem of *path explosion* [18, 78]. A naive and exhaustive program analysis is not scalable because the number of all possible path combinations can grow exponentially to the size of the applications. Hence, our taint and string analysis is aborted after a threshold. By using intermediate caches for the analysis results, the amount of paths is reduced for the overlapping analysis of previously explored paths.

Next to the general limitations of SCA, our approach comes with certain design specific limitations. The detection of path-sensitive security mechanisms in our approach is unsound: although our approach detects basic path-sensitive sanitization and termination,

it does not succeed in analyzing inter-procedural path-sensitive security mechanisms. For example, path-sensitive input validation or sanitization within a user-defined function, whose utilization depends on a function parameter, can lead to false positives. Due to the usage of function summaries, all possible return values are considered independently of the call-site's arguments. As introduced in Section 3.6.2, this also affects the receiver analyses of invoked methods on an object passed as an argument. Additionally, infeasable path combinations can lead to false positives. While these limitations are specific to our approach and can be possibly corrected with other techniques, such as constraint solvers [137,160], a great loss in performance is likely that would let an extended approach not scale to large applications. Thus, only best-effort techniques are in place that can fail under certain circumstances. Last but not least, our approach does not yet consider all language features, such as *traits* and *references*, but can be extended in the future in a straightforward way.

3.11 Discussion

In this chapter, we presented the design of a novel static code analysis tool that aims to automatically and accurately detect taint-style vulnerabilities in PHP applications. It models the dynamic PHP language by using symbolic execution and abstract interpretation and it supports 45 different scopes in 36 different vulnerability types with fine-grained context-sensitive taint analysis. Block and function summaries enable a very efficient data flow analysis that scales to large applications. The key is a very precise block simulation that emulates and summarizes the effects of built-in language features, including complex object-oriented code, in reuseable summaries. The efficient data flow through weakly typed variables, arrays, and properties is enabled by a lightweight intermediate representation. It infers data symbols and tags from the block summaries in combination with a forwards-directed propagation of object data between basic blocks and is used to resolve dynamic data. Moreover, our approach statically models the data flow through persistent data stores by collecting all storage writings and readings. At the end of the analysis, we can determine if data read from a persistent store can be controlled by an attacker and if this leads to a second-order vulnerability. Our inter-procedural and field-sensitive analysis allows us to generate POP chains in the context of a POI vulnerability.

While our intermediate representation does not enable a sound heap analysis, we believe that a fair tradeoff between precision and scalability for security analysis is achieved. Previous research has shown that sound heap, type, and value analysis is not efficient for the security analysis of large applications [53,149]. Additionally, we favor a backwards-directed taint analysis of sensitive sinks over a forwards-propagation of all tainted sources [61,66]. Considering a long analysis path between a source and a sink, our intuition is that a security mechanism is eventually applied closer to the sink than to the source in PHP applications (e. g., a typecast before a SQL query). Hence, a backwards-directed taint analysis can be aborted earlier at the security mechanism than a forwards-directed analysis. Redundant analyses are prevented by caching the analysis results. In Section 3.10, we discussed limitations of our approach which stem from the nature of static analysis, as well as from our tradeoffs. The evaluation of a prototype implementation in the next chapter will show if our approach is effective in practice.

Chapter 4

Evaluation of a Prototype Implementation

We implemented our approach for the static security analysis of PHP code in a prototype. In this chapter, we evaluate the prototype's analysis results for popular and large real-world applications. The benefits of our tool are measured by its capabilities to model the programming language, to analyze the model correctly, and to precisely detect different types of vulnerabilities. The *precision* (`TP/(TP+FP)`), *recall* (`TP/(TP+FN)`), and *false discovery rate* (`FP/(TP+FP)`) indicate how well our tool performs. Moreover, the performance of our tool is measured by the consumed time and memory. Our evaluation shows that our approach is capable of efficiently analyzing modern PHP applications and can detect complex security vulnerabilities previously unknown. It is split into three experiments.

In Section 4.1, we evaluate the ability to detect different types of taint-style vulnerabilities with five applications. In total, we analyzed almost half a million LOC and found that on average every 4th line required taint analysis and that our prototype covers 89 % of all called built-in functions. Overall, we detected and reported 73 previously unknown vulnerabilities with a precision of 72 % and a recall of 88 %. Addtionally, we analyzed two web applications that were used during the evaluation of prior work in this area and found that we outperform existing tools by finding 31 vulnerabilities previously missed.

In Section 4.2, we evaluate the unique ability of our prototype to detect second-order vulnerabilities with six applications. Our prototype covered 71 % of all available PDS in an overall of 143 KLOC. It reported a total of 159 valid second-order vulnerabilities with a precision of 79 % (recall 95 %). We detected various *remote code execution* vulnerabilities and our evaluation revealed that second-order vulnerabilities are highly critical.

In Section 4.3), we evaluate the detection of POI vulnerabilities and our novel algorithm for the construction of POP gadget chains. In 10 popular applications recently affected by POI with a total of 1.73 million LOC, we detected 30 novel POI vulnerabilities and 28 novel POP chains with a precision of 84 %. With the help of our novel chains, the severity of 6 known POI vulnerablities in our 10 selected applications was refined.

The experiments were performed on a machine with an Intel i7-2600 CPU @ 3.40 GHz and 16 GB RAM. In total, we analyzed 23 different applications, 9 555 files, and 2.38 million *lines of code* (LOC). We detected 321 previously unknown vulnerabilities with an average false discovery rate of 20 % (discussed in Section 4.4) and reported all issues to the vendors in a responsible way.

4.1 Taint-style Vulnerabilities

We evaluated our tool's precision to detect taint-style vulnerabilities with the following five popular open source applications: *HotCRP 2.60* [75], *MyBB 1.6.10* [91], *osCommerce 2.3.3* [96], *phpBB2 2.0.23*, and *phpBB3 3.0.11* [100]. In total, our prototype detected 73 previously unknown security vulnerabilities, mainly XSS, SQLi, and file related issues. A high-level overview of the vulnerability distribution among the different types is listed in Table 4.1. The vulnerability distribution among the different applications is listed in Table 4.2, next to the overall number of analyzed PHP files and LOC. Furthermore, we enumerated the number of sensitive sinks that required taint analysis (TA). The maximum memory peak (MP) is denoted in *megabytes* and the overall scan time (ST) is denoted in *seconds* in Table 4.2.

In the following, we first introduce our performance results (see Section 4.1.1). Then, we evaluate the built-in function coverage of our tool in Section 4.1.2. We highlight the most interesting true positive findings in Section 4.1.3 and discuss false positives of our prototype in Section 4.1.4. In order to evaluate false negatives, we scanned old versions of the selected applications that are affected by known vulnerabilities (see Section 4.1.5). Such an evaluation approach enables us to estimate which vulnerabilities our prototype was not able to find in an automated way. Finally, we analyzed two PHP projects that were evaluated by other researchers working in this area and we directly compare our approach against other tools in Section 4.1.6.

Table 4.1: Detected vulnerability types

Type	TP	FP	FN
Cross-Site Scripting	48	22	5
SQL Injection	11	7	4
Arbitrary File Write	8	0	0
Path Traversal	3	0	0
Variable Tampering	2	0	0
HTTP Response Splitting	1	0	1
Total	73	29	10

Table 4.2: Evaluation results for popular real-world applications.

Software	Files	LOC	TA	TBC	TBI	UBC	UBI	MP	ST	TP	FP	FN	CVE
HotCRP	72	39 938	19 420	5 171	289	170	51	293	55	7	4	0	0
MyBB	327	138 357	55 917	8 152	1 287	225	115	1 117	188	2	0	8	10
osCommerce	545	65 556	7 453	9 059	860	184	85	476	60	48	19	1	29
phpBB2	176	46 287	10 623	3 666	340	144	56	289	29	13	6	1	2
phpBB3	270	186 814	43 616	7 554	1 273	269	192	1 143	252	3	0	0	1
Total	1 390	476 952	137 029	33 602	4 049	676	294	3 318	584	73	29	10	42
Average	278	95 390	27 406	89%	11%	70%	30%	664	117	72%	28%	24%	8

4.1.1 Performance

We scanned a total of 1 390 PHP files with almost half a million LOC. On average, every 4th line of code contained a sensitive sink that required taint analysis. The average memory peak usage per project was 664 MB and the average scan time was about 2 minutes. In other words, our prototype needed 7 MB and 1.23 seconds per KLOC. The largest evaluated software *phpBB3* with over 186 000 LOC had a scan time of less than 5 minutes and required a bit more than 1 GB of memory. Thus, we are positive that our approach scales to even larger projects.

4.1.2 Built-in Function Coverage

In order to evaluate the built-in function coverage of our tool, we logged the name of every inspected function. In case a function was not user-defined within the software, it was considered as a built-in function of PHP. We then examined if the function name is covered by our tool or was ignored during analysis. Table 4.2 shows the *total* number of *built-in* functions our tool *covered* (TBC) or *ignored* (TBI). Furthermore, we enumerated the *unique* number of *built-in* functions our tool *covered* (UBC) or *ignored* (UBI).

On average, every 13th line of code contains a built-in function call, excluding the lines that call user-defined functions (which can lead to further built-in function calls). The high amount of built-in function usage emphasizes the need for precise function simulation during code analysis. Within our five analyzed applications, a call to 970 *unique* built-in functions was detected. Our tool simulates 70% of these unique functions which covers 89% of all defined calls within the applications. The remaining calls which are ignored by our tool are mainly related to database, image, or sort functions and — to the best of our knowledge — these functions do not affect the overall taint analysis results.

4.1.3 True Positives

In total, 72 % of the reported issues in our first evaluation are true positives. A true positive was counted for every vulnerable line of code. This means that a vulnerability inside a function was counted only for once, and only if the function was called in an exploitable context and not for every call. Sometimes, a valid report was counted even if the vulnerability is not exploitable. For example, the same input could be used in two differently constructed SQL queries but the application exits after the failure of one SQL query. Although then it is not possible to craft an injection that fits both SQL queries, two valid reports were counted nonetheless. In this case, fixing only the first SQL query would allow to exploit the second SQL query and thus both reports are important. We now examine selected vulnerabilities in three different projects in order to illustrate their complexity and severity. It is evident that these vulnerabilities could only be detected with our novel approach of precisely simulating different language features and their interaction.

phpBB2

phpBB is a well-known open source bulletin board software [100]. It is developed in the current version *phpBB3*, however, its predecessor *phpBB2* is still widely used and

also integrated into popular software, such as *PHP-Nuke* [99]. In total, our prototype reported 13 vulnerabilities in the latest *phpBB2* version 2.0.23. The vulnerabilities also affect the latest *PHP-Nuke* version. Our tool detected six rather harmless SQL injection vulnerabilities in the installer based on a user-supplied database table prefix. Additionally, two critical SQL injection vulnerabilities in the administration interface were detected. The simplified code of one of these SQL injections is shown in Listing 4.1.

```
1  $style_name = urldecode($_GET['style']);
2  $install_to = urldecode($_GET['install_to']);
3  $template_name = $$install_to;
4  for($i = 0; $i < count($template_name); $i++) {
5     if($template_name[$i]['style_name'] == $style_name) {
6        while(list($key, $val) = each($template_name[$i])) {
7           $db_fields[] = $key;
8           $db_values[] = addslashes($val);
9        }
10    }
11 }
12 $sql = "INSERT INTO " . THEMES_TABLE . " (";
13 $sql .= implode(',', $db_fields);
14 $sql .= ") VALUES (";
15 $sql .= "'" . implode("','", $db_values) . "'";
16 $sql .= ")";
17 mysql_query($sql);
```

Listing 4.1: Simplified code of a SQL injection in *phpBB2*.

In line 3, a *variable* variable based on unsanitized user input is assigned to the variable `$template_name`. The application assumes that `$template_name` is provided as an array which lists several templates. First, it loops through all elements of `$template_name` and compares the `style_name` of the template with the provided GET parameter *style*. If the specified template was found, the application saves the template's array key names to the array `$db_fields` (line 7) and all array values sanitized to the array `$db_values` (line 8). Then, all `$db_fields` are used as column identifiers in a SQL `INSERT` query in line 13 and all `$db_values` are used as the values to be inserted. A vulnerability occurs because the `$db_fields` are not sanitized but can be influenced by an attacker. For exploitation, the *install_to* parameter is set to *_GET* such that the variable `$template_name` points to the GET parameters controlled by the user. Then, the SQL injection can be exploited as shown in Listing 4.2. The vulnerability is not present in the *phpBB3* code base.

```
admin_styles.php?style=rips&install_to=_GET&0[style_name]=rips
   &0[template_name)VALUES('sqli','sqli')-- -]=1
```

Listing 4.2: SQL injection exploitation through an array key.

The rather complicated code demonstrates the importance of simulating PHP's built-in features. First of all, the data flow through several built-in functions such as `urldecode()`, `list()`, `each()`, and `implode()` has to be analyzed precisely. The challenge is to model the array handling of these functions. If one of these functions is not or imprecisely simulated, then the vulnerability is not detected. Moreover, we encountered a *variable* variable in line 3 and a *while* loop in line 6 which require analyses of *variable* elements. Finally, sanitization is applied in line 8 but not in line 7 and the SQL query requires context-sensitive string analysis to decide whether the sanitization is sufficient or not.

HotCRP

HotCRP is a popular conference management software that is used by several top tier conferences. Our current prototype reported 7 XSS and 4 SQLi vulnerabilities in the latest version 2.60. In six out of seven reported XSS vulnerabilities a user supplied parameter is reflected unsanitized to the HTML response page and a true positive was reported.

Moreoever, one out of four reported SQLi vulnerabilities is a true positive. It affects an `INSERT` query where a new paper can be added by an unprivileged user. Because *error reporting* is implemented for SQL queries and the users' passwords are stored in plaintext in the database, an attacker can easily read the conference administrator credentials (see Figure 4.1). This enables an attacker to compromise the conference administration account and to review, edit, delete, or accept submitted papers of her choice.

```
1 $v = defval($_REQUEST, "emailNote", "");
2 echo "<input type='text' name='emailNote' size='30' value='",
3    htmlspecialchars($v=="" ? "Optional explanation" : $v),
4    "' />";
```

Listing 4.3: Weak output sanitization in *HotCRP*.

An XSS vulnerability shown in Listing 4.3 demonstrates our ability to detect weak sanitization. The user-defined function `defval()` returns user input that is embedded into the HTML page. The user input is sanitized with the built-in function `htmlspecialchars()` in line 3, however, the second parameter is not specified in order to escape single quotes (see Section 2.3.2). Previous work would miss this vulnerability because `htmlspecialchars()` is handled context-insensitive as valid sanitization method.

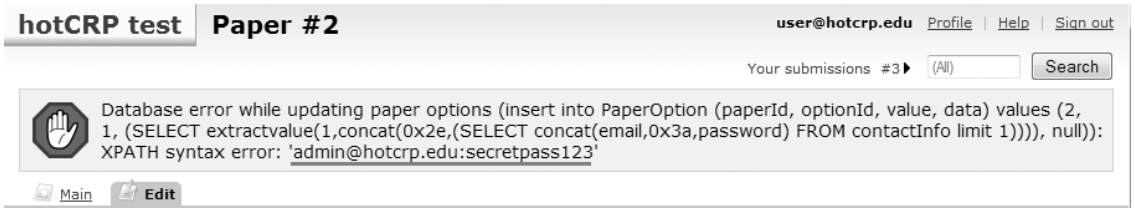

Figure 4.1: A SQL injection in HotCRP allows to leak the administrator's password to an unprivileged user in plaintext via SQL error message.

osCommerce

osCommerce is a popular online store software that allows to sell products and services. We were able to identify 48 vulnerabilities in the latest version 2.3.3. Our prototype detected a SQL injection vulnerability in the installer and in the administration interface. Combined with an XSS vulnerability, the second SQLi vulnerability allows an unprivileged attacker to retrieve the administrator's password hash by sending a malicious link to an administrator. Additionally, 40 XSS vulnerabilities were detected in the administrator interface and in the installer. The root cause is shown in Listing 4.4.

```
1 $HTTP_GET_VARS = array_map('addslashes', $_GET);
2 echo '<tr onclick="document.location.href=\'' . BASE_URL.
3 'page=' . str_replace('&amp;', '&', htmlspecialchars($HTTP_GET_VARS['page'])) . '\'">';
```

Listing 4.4: An XSS vulnerability in eventhandler context.

The GET parameter *page* is used in the eventhandler `onclick` of a table row. First, it is not possible to break out of the outer double quotes of the eventhandler because `htmlspecialchars()` is used. Second, although the parameter `ENT_QUOTES` is not set to encode single quotes, it is not possible to break out of the inner single quotes in the JavaScript code because *osCommerce* uses the function `addslashes()` for each user supplied parameter that adds a preceding backslash to each single quote. Third, it is not possible to inject a `javascript:` protocol handler because the constant `BASE_URL` is prefixed to the new location.

However, our tool validly reported an XSS vulnerability because the injection context is an eventhandler. Here, the browser interprets HTML entities within the JavaScript code as their original character representation. Thus, we are able to inject the HTML entity `&#39;` to break out of the inner single quotes and to inject our own JavaScript code. The attack is demonstrated in Listing 4.5 with an *urlencoded* payload that is triggered if the user clicks on the table row. Again, the vulnerability stresses the importance of context-sensitive string analysis and the correct handling of PHP's built-in functions.

```
admin/customers.php?page=%26%2339%3B-alert(1)-%26%2339%3B
// &#39;-alert(1)-&#39;
```

Listing 4.5: Urlencoded and decoded payload for an XSS exploit.

Additionally, our tool reported various file vulnerabilities in the installer and the administration interface, for example, a *file write* vulnerability that allows to write arbitrary PHP code into language files which leads to *remote code execution*. Although we counted these as valid reports, the affected code represents a feature and is not interpreted as a security vulnerability by the developers.

Furthermore, an XSS is reported for each SQL query that contains user input and could result in an error because the SQL query is printed unsanitized within the error handler. Although these reports are valid and were successfully verified with the present SQL injection vulnerability, we ignored these issues and did not count them as true positive. Contrarily, they could be counted as false positive because the SQL queries do not fail in typical situations.

4.1.4 False Positives

In total, 28 % of the reported vulnerabilities in our first evaluation turned out to be false positives. The root causes for these invalid reports are:

- Path-insensitive data flow analysis
- Undetected sanitization through a database whitelist
- Wrong content-type detection

The root cause for 19 false positives in *osCommerce* is shown in Listing 4.6. Here, a user-defined function sanitizes its first argument based upon the second argument. Because our prototype performs path-insensitive data flow analysis and is based on function summaries, it wrongly integrates both possible `return` values into the function summary that is then used, regardless of the second argument.

```
1 function tep_output_string($string, $protected = false) {
2    if ($protected == true) {
3       return htmlspecialchars($string);
4    } else {
5       return $string;
6    }
7 }
```

Listing 4.6: The root cause for false positives in *osCommerce*.

In *HotCRP* an XSS vulnerability was reported erroneously. Here, a user-supplied email address is printed to the HTML page unsanitized, however, the email address is checked for presence in the database first. Because the format is checked before a new email address is added to the database, the email address is sanitized indirectly. Furthermore, three reported SQLi vulnerabilities are false positives. Our prototype was unable to detect path-sensitive sanitization of tainted values [36, 159].

Another reported XSS vulnerability was counted as false positive. Although user input is printed unsanitized to the HTML page, the vulnerability is not exploitable because the HTML response header `content-type` is changed to `text/plain`. Thus, a browser will not render injected HTML and XSS attacks are prevented.

4.1.5 False Negatives

Evaluating false negatives is a difficult task because the number of existing vulnerabilities in a software is unknown. To obtain an estimated result, we collected all CVE entries from the CVE database [88] which are related to injection flaws in our selected PHP projects. We then run our tool against the affected versions of the software and searched for a vulnerability report that matches the CVE details. During this process, we encountered the following obstacles. First of all, no CVE entries exist for *HotCRP*. Secondly, only very few CVE entries for *phpBB* are relevant because most of them describe vulnerabilities in external plugins. For *MyBB* and *osCommerce* a fair amount of CVE entries is available but certain old versions of *MyBB* are unavailable on the Internet.

In total, we examined 42 CVE entries in 7 different software versions. Our tool correctly identified 32 of the described vulnerabilities in an automated way, resulting in an estimated recall of 76 %. However, if we exclude *MyBB*, the recall for the rest of our selected applications is 95 %. The root causes for false negatives are:

- field-insensitive data flow analysis
- second-order vulnerabilities

Our tool missed 8 out of 10 vulnerabilities in *MyBB* because it did not fully support analysis of object-oriented code at the time of our first evaluation. All false negatives in *MyBB* are based on the same problem: our prototype misses the data flow of GET and POST parameters because they are written to and retrieved from object fields. Two other false negatives stemed from the fact that our tool did not handle the data flow through externally stored data at the time of our first experiment. Thus, it misses second-order vulnerabilities such as *Persistent XSS*. We later added support for these features which will be evaluated in Section 4.2 and 4.3.

4.1.6 Comparison

In previous work on static analysis of PHP applications, several evaluation results for different software applications were reported [68,149,154]. Comparing our results to previous work is not straightforward for several reasons. First of all, often we have no access to the implemented prototype. Second, we do not know exactly how the amount of detected vulnerabilities was counted, and as discussed in Section 4.1.3 this is a hard problem itself. As a result, comparing only the numbers of found true and false positives may be misleading.

For a better approach, we chose to evaluate software that was analyzed by other researchers with the following criteria: (1) the software is still available on the Internet, (2) there is a follow-up version, and (3) the follow-up version introduces security patches and does not add new main features. We can then compare our results to the stated results in previous work for the exact software version, but more importantly, we can assume that any vulnerability we detect in the follow-up version was missed by previous work.

The software *NewsPro* [144] and *myBloggie* [92] match our criterias and was evaluated by Jovanovic et al. [68] and by the work of Xie and Aiken [154]. Pixy supports the detection of XSS and SQLi vulnerabilities, while the prototype of Xie and Aiken only detects SQLi vulnerabilities. Our results compared to the others are listed in Table 4.3. The total precision of 98 % and false discovery rate of 2 % stem from the fact that the code of both software is relatively small and simple compared to our selected real-world applications. This is shown by the LOC as well as the *total* (TB) and *unique* (UB) amount of *built-in* functions used. In the following, we discuss our findings for *NewsPro* and *myBloggie*.

Table 4.3: Compared evaluation results for previously studied real-world applications.

						Our prototype				Jovanovic et al.				Xie & Aiken	
						XSS		SQLi		XSS		SQLi		SQLi	
Software		Files	LOC	TB	UB	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
NewsPro	1.1.4	23	5 047	827	56	5	0	18	0	4	14	14	34	8	0
NewsPro	1.1.5	23	5 077	841	57	4	0	6	0	-	-	-	-	-	-
myBloggie	2.1.3b	91	11 487	1 218	122	15	0	26	3	13	3	31	11	16	0
myBloggie	2.1.4	92	11 772	1 235	124	13	0	8	0	-	-	-	-	-	-
Total		229	33 383	4121	134	37	0	58	3	17	17	45	45	24	0
Average		57	8 346	1030	90	100%	0%	95%	5%	50%	50%	50%	50%	100%	0%

NewsPro

Utopia NewsPro is a news management system and we evaluated version *1.1.4* and the follow-up version *1.1.5*. Our prototype reported 5 XSS vulnerabilities in version *1.1.4* whereas Pixy reported 4 XSS vulnerabilities. According to the CVE details [88], the XSS vulnerabilities reported by Pixy mainly base on the deprecated *register_globals* setting, which is disabled for our prototype by default. Because the follow-up version *1.1.5* contains 4 of our 5 XSS vulnerabilities, we conclude that Pixy missed these issues. Furthermore, Pixy had a false discovery rate of 77% while our prototype reported no false positives in the *NewsPro* application.

Additionally, our tool reported 18 SQLi vulnerabilities with no false positives. The prototype of Xie and Aiken reported only 8 SQL injections and seem to miss certain vulnerabilities. Even with *register_globals* enabled, which introduces far more security issues, Pixy detected only 14 SQLi vulnerabilities with a false discovery rate of 71%. We believe that our prototype detected more SQL injections compared to prior work, aided by the fact that 6 of our detected SQL injections are still present in the follow-up version *1.1.5*.

myBloggie

We evaluated the weblog system *myBloggie 2.1.3 beta* and its successor *myBloggie 2.1.4*. According to the authors, Pixy reported 13 XSS vulnerabilities in *myBloggie 2.1.3 beta*. Because 13 out of our 15 detected XSS vulnerabilities in *myBloggie 2.1.3 beta* are still present in the follow-up version, we assume that we identified different vulnerabilities than Pixy. A closer look at the released advisory reveals that Pixy reported issues based on the deprecated PHP setting *register_globals* [65]. Our vulnerabilities base on the source `$_SERVER['PHP_SELF']` which is not modeled by Pixy. In Section 6.2 of their work it is wrongly stated that "the predefined PHP variables `$_SERVER['PHP_SELF']` and `$_SERVER['HTTP_HOST']` are untainted, since they cannot be controlled by an attacker" [67]. Pixy encountered 3 false positives whereas our prototype only reported true positive XSS vulnerabilities.

Furthermore, our prototype reported 26 SQLi vulnerabilities. Three false positives occured due to path-sensitive sanitization. Because the prototype of Xie and Aiken is path-insensitive as well but did not encounter these false positives, we conclude that our prototype analyzed more data flow. Their prototype reported only 16 SQLi vulnerabilities which supports this assumption.

Pixy detected 31 SQLi vulnerabilities in *myBloggie*. Because 8 of our 26 detected SQLi flaws are still present in the follow-up version, we approximate that Pixy detected only 18 of the 26 SQLi vulnerabilities. Another 13 SQLi vulnerabilities probably base on the *register_globals* setting. Furthermore, 36% of the SQLi vulnerabilities in *myBloggie* reported by Pixy are false positives.

Finally, we do not know if one or more vulnerabilities was detected by Pixy or the prototype of Xie and Aiken but was missed by our prototype. A rather complicated XSS vulnerability in *myBloggie 2.1.3 beta* described in detail by Jovanovic et al. [68] was detected by our tool.

4.2 Second-Order Vulnerabilities

In a second experiment, we added the ability of detecting second-order vulnerabilities to our prototype and evaluated it with six real-world web applications. We chose the conference management systems *OpenConf 5.30* and *HotCRP 2.61* for their popularity in the academic field, and *osCommerce 2.3.3.4* for its large size. Furthermore, we evaluated the follow-up versions of the most prominent software used in related work [6,68,154,158]: *NewsPro 1.1.5*, *MyBloggie 2.1.4*, and *Scarf 2007-02-27*. A summary of our results for these selected applications is shown in Table 4.4.

Our prototype reported a total of 159 valid second-order vulnerabilities with a false discovery rate of 21 %. In summary, 97 % of the valid reports are *persistent XSS* vulnerabilities where the payload is stored in the database. Five *persistent XSS* vulnerabilities are caused by session data or by file names. This is closely related to the fact that 94 % of all taintable PDS we identified are columns in database tables (see Section 4.2.2) and that sensitive sinks, such as `echo`, are one of PHP's most prominent built-in features [55]. Among our second-order vulnerability reports were 14 valid multi-step exploits. As shown in Table 4.10, these base on 20 detected SQL injections and two file create vulnerabilities.

In this section, we evaluate the performance overhead of our implementation for the detection of second-order vulnerabilities (see Section 4.2.1). We then evaluate the amount of PDS in our selected applications and the coverage of our tool in Section 4.2.2. Furthermore, we examine our detected critical second-order vulnerabilities and multi-step exploits in Section 4.2.3. Finally, in Section 4.2.4 and 4.2.5, we discuss false positives and negatives of our prototype.

Table 4.4: Our evaluation results for selected applications.

Software	Files	LOC	TP	FP	FN
osCommerce	570	66 381	97	29	6
HotCRP	74	40 339	1	1	0
OpenConf	121	20 404	16	4	0
NewsPro	23	5 077	7	1	0
Scarf	19	1 686	37	8	3
MyBloggie	58	9 485	1	0	0
Total	865	143 372	159	43	9
Average	144	23 895	79%	21%	

4.2.1 Performance

We evaluated our prototype implementation *with* the ability to detect second-order vulnerabilities (+SO) and *without* (-SO). The amount of memory consumption (M, in *megabytes*), scan time (T, in *seconds*), and second-order vulnerability reports (R) for our selected applications are given in Table 4.5.

While the memory consumption does not increase significantly by adding second-order analysis, the average scan time increases by 40 %. Note, however, that this includes

217 processed vulnerability reports the prototype would have missed without the additional second-order analysis. Furthermore, we believe that a total scan time of less than 11 minutes for our six selected applications with a total of 143 KLOC is still reasonable.

Table 4.5: Performance results for selected applications.

	-SO Analysis		+SO Analysis		
Software	M[mb]	T[s]	M[mb]	T[s]	R
osCommerce	834	134	846	213	129
HotCRP	752	186	775	345	3
OpenConf	528	33	523	47	21
NewsPro	50	1	50	3	17
Scarf	39	1	40	14	46
MyBloggie	87	7	87	11	1
Total	2290	362	2321	633	217
Average	382	60	387	106	36

4.2.2 PDS Usage and Coverage

In order to obtain an overview of the usage of PDS in web applications, we manually evaluated the total amount of different memory locations. Note that these numbers do not reflect how often one memory location is used at runtime. Then, we evaluated the ability to taint these memory locations by an application's user and compared it to the detection rate of our prototype. A PDS is defined as *taintable* if it can contain at least one of the following characters submitted by an application user: `\<>'"`. In total, we manually identified 841 PDS of which 23 % are taintable. Our prototype successfully detected 71 % of the taintable PDS with a false discovery rate of 6 %. In the following, we present our results for different types of PDS.

Databases

Our implementation successfully recovered the database schema for all tested applications during the initialization phase. For evaluation, we categorized all available columns in the application's database schema by their declared data type. Only columns with a *string* type, such as `VARCHAR` or `TEXT`, are of interest because these can store attack payloads. As shown in Table 4.6, we found that on average about half of the columns are not taintable due to numeric data types, such as `INT` and `DATE`.

We then carefully fuzzed a local instance of each application manually with common attack payloads in order to determine which columns of type *string* are taintable. Furthermore, we observed which columns were reported by our prototype implementation as taintable when the schema is available and when not. The results are compared in Table 4.7. Among the columns with a string type, 53 % are taintable. As a result, only 24 % of all available columns are not sanitized by the application or the columns' data type.

For the rather old and simple applications, all taintable columns were detected by our prototype. The modern and large applications often use loops to construct dynamic SQL queries where reconstruction is error-prone. Overall, we detected 70 % of all taintable

columns. In case the database schema is known, only 5 % of our reports are false positives. The root cause is path-sensitive sanitization of data that is written to the database—a sanitization that our current prototype is not able to detect yet. The false discovery rate is higher if the database schema of an application is not found. In this case, a static analysis tool cannot reason about data types within the database and could flag columns with a numerical data type as taintable.

Table 4.6: Detected column types.

Software	Tables	Columns	Num	String
osCommerce	50	331	193	138
HotCRP	29	217	142	75
OpenConf	18	129	48	81
NewsPro	8	43	18	25
Scarf	7	37	22	15
MyBloggie	4	24	10	14
Total	116	781	55%	45%

Table 4.7: Detected taintable columns.

		Schema		No schema	
Software	Tainted	TP	FP	TP	FP
osCommerce	63	55	4	55	37
HotCRP	43	27	1	27	3
OpenConf	47	16	1	16	4
NewsPro	12	12	0	12	0
Scarf	10	10	1	10	3
MyBloggie	9	9	0	9	0
Total	184	70%	5%	70%	27%

Sessions

To obtain a ground truth for our evaluation, we again manually assessed the applications' code for all accessed keys of the superglobal `$_SESSION` array. Dynamic keys were reconstructed and keys in multi-dimensional arrays were counted multiple times. Then, we manually examined which session keys are taintable by the application's user and compared them to the analysis result generated by our prototype implementation. As shown in Table 4.8, we found that only 12 % of the 52 identified session keys are taintable within our selected applications.

Our prototype correctly detected all taintable session keys. One FP occurred because the sanitized email address of a user is written to the session after it is fetched from the database. This FP is based on the previously introduced FP in identifying taintable columns. A custom session management in *osCommerce* could not be evaluated with our approach and led to the exclusion from our evaluation.

Table 4.8: Detected taintable session keys.

Software	Keys	Taintable	TP	FP
osCommerce	41	-	0	0
HotCRP	29	2	2	0
OpenConf	14	2	1	0
NewsPro	2	1	1	0
Scarf	4	0	0	1
MyBloggie	3	1	1	0
Total	52	12%	83%	16%

Table 4.9: Detected taintable path names.

Software	Paths	Taintable	TP	FP
osCommerce	2	2	2	0
HotCRP	1	0	0	0
OpenConf	1	0	0	1
NewsPro	1	0	0	0
Scarf	1	1	1	0
MyBloggie	2	2	2	0
Total	8	63%	100%	16%

File Names

In order to evaluate the features that allow an application's user to alter a file name, we manually assessed each application for *file upload*, *file creation*, and *file rename* features and enumerated the different target paths in order to obtain a ground truth. Next, we compared them to the taintable path names that were reported by our prototype. The results are shown in Table 4.9. We found at least one feature in each of the application's source code to create a new file. However, half of the applications sanitize the name of the file before creating it. Our prototype detected all taintable path names. One FP occurred for *OpenConf*, where uploaded files are sanitized in a path-sensitive way.

Interestingly, a file upload in *Scarf* is based on a second-order data flow. The name of the uploaded file is specified separately and stored as a configuration value in the database before it is read from the database again and the file is copied. Because no sanitization is applied, an administrator is able to copy any file to any location of the server's file system which leads to *remote code execution*. This critical vulnerability was missed in previous work that also used this application for evaluating their approach [6, 158].

4.2.3 True Positives

Our evaluation revealed that second-order vulnerabilities are highly critical. Next to *persistent XSS* and file vulnerabilities, we detected various *remote code execution* vulnerabilities in *osCommerce*, *OpenConf*, and *NewsPro*. In the following, we introduce two selected vulnerabilities in order to illustrate the complexity and severity of real-world second-order vulnerabilities. It is evident that these vulnerabilities could only be detected with our novel approach of analyzing second-order data flows.

Our prototype reported two *arbitrary file upload* vulnerabilities and 14 SQL injection vulnerabilities. Because these vulnerabilities affect a storage operation, the stored data can be manipulated during multi-step exploitation. Our prototype found 14 valid multi-step exploits with a single FP, as shown in Table 4.10. All detected multi-step exploits consist of two steps and no *third-order* vulnerabilities were detected within our selected applications. In the following, we examine two multi-step exploits in *osCommerce* that lead to *remote command execution*. These vulnerabilities could only be detected with our novel approach of analyzing multi-step exploits.

Table 4.10: Detected vulnerability types.

Software	File TP	SQLi TP	SQLi FP	Multi-Step TP	Multi-Step FP
osCommerce	1	3	0	3	0
HotCRP	0	1	7	0	1
OpenConf	0	4	1	1	0
NewsPro	0	6	0	9	0
Scarf	1	1	0	1	0
MyBloggie	0	5	0	0	0
Total	2	20	8	14	1
Average	100%	71%	29%	93%	7%

Second-Order LFI to RCE in OpenConf

OpenConf is a well-known conference management software used by many (academic) conferences. Our prototype detected a *second-order local file inclusion* vulnerability in the user-defined `printHeader()` function that leads to *remote command execution*. The relevant parts of the affected file *include.php* is shown in Listing 4.7.

```
1  function printHeader($what, $function="0") {
2     require_once $GLOBALS['pfx'] .
3        $GLOBALS['OC_configAR']['OC_headerFile'];
4  }
5
6  $r = mysql_query("SELECT `setting`, `value`, `parse` FROM `" . OCC_TABLE_CONFIG . "`");
7
8  while ($l = mysql_fetch_assoc($r)) {
9     $OC_configAR[$l['setting']] = $l['value'];
10 }
11 printHeader();
```

Listing 4.7: Simplified *include.php* of *OpenConf*.

When looking at the code, it does not reveal any vulnerability. Whenever the code is included, settings are loaded from the database and the user-defined function `printHeader()` is called. This function includes a configured header file and prints HTML code.

```
1  function updateConfigSetting($setting, $value) {
2     $q = "UPDATE `" . OCC_TABLE_CONFIG . "`
3           SET `value`='" . safeSQLstr(trim($value)) . "'
4           WHERE `setting`='" . safeSQLstr($setting) . "'";
5     return(ocsql_query($q));
6  }
7
8  foreach (array_keys($_POST) as $p) {
9     if (preg_match("/^OC_[\w-]+$/", $p)) {
10       updateConfigSetting($p, $_POST[$p]);
11    }
12 }
```

Listing 4.8: Simplified code to change settings in *OpenConf*.

However, as shown in Listing 4.8, it is possible for a privileged chair user to change any configuration setting. The configuration page does not specify an input field to change the *headerFile* setting. Nonetheless, by adding the key *OC_headerFile* to a manipulated HTTP POST request, the setting is changed. The loop over the submitted keys of the superglobal `$_POST` array in line 8, as well as the loop over the `$OC_configAR` in line 9, demonstrate the importance of tracking the taint status of PHP's array keys precisely. At this point, a chair member can include any local file of the system to the output. Additionally, because the software allows to upload PDF files to the server, our prototype added a multi-step exploit report. Indeed, if a PDF file that contains PHP code is uploaded by an attacker to the server and the *headerFile* setting is pointed to that PDF, then the attacker's PHP code is executed. Moreover, our tool reported a SQL injection vulnerability that is accessible to unprivileged users. This allows any visitor to extract the chair's password hash (salted SHA1) from the database.

Second-Order RCE in NewsPro

Utopia NewsPro is a software for blogging and it was used in previous work for evaluation [150, 154, 158]. Our prototype reported a *second-order code execution* vulnerability in the administrator interface. Here, a user is able to alter the template files of the blog. The simplified code is shown in Listing 4.9.

```
1 $tempid = (int)$_POST['tempid'];
2 $template = mysql_real_escape_string($_POST['template']);
3 $query = "UPDATE `unp_template` SET template='$template' WHERE id='$tempid'";
4 $updateTemplate = mysql_query($query);
```

Listing 4.9: Simplified code to change the template in *NewsPro*.

The template code is read from the database in various places of the source code with the help of the user-defined function `unp_printTemplate()` (see Listing 4.10). First, this function writes the template's code to a cache array (line 6) and then returns it from this array again. The example demonstrates the importance of inter-procedural analysis and array handling.

```
1  function unp_printTemplate($template) {
2     global $templatecache, $DB;
3     $getTemplate = mysql_query("SELECT name,template
4       FROM `unp_template` WHERE name='$template' LIMIT 1");
5     while ($temp = mysql_fetch_array($getTemplate)) {
6        $templatecache[$template] = $temp['template'];
7     }
8     return addslashes($templatecache[$template]);
9  }
10 eval('$headlines_displaybit = "' . unp_printTemplate('headlines_displaybit') . '";');
```

Listing 4.10: Simplified *Remote Code Execution* vulnerability in *NewsPro*.

At the call-site, in line 10, the fetched template is evaluated with PHP's `eval` operator that executes PHP code. The template's code is escaped (line 8), however, the double-quoted value of the evaluated variable `$headlines_displaybit` allows to execute arbitrary PHP code using curly syntax. By adding the code `{${system(id)}}` to a template, the system command *id* is executed. Note that related work missed to detect this vulnerability, which is also present in prior versions.

Multi-Step RCE in osCommerce

OsCommerce is a popular e-commerce software. For one of three reported SQLi vulnerabilities in *osCommerce*, our prototype additionally reported a *multi-step remote code execution* exploit. The SQLi is located in the backup tool of the administrator interface and shown in Listing 4.11. Here, a SQL file is uploaded to restore a database backup. Since the name of the uploaded file is later used unsanitized in a SQL query (line 5), an attacker is able to insert any data into the `configuration` table by uploading a SQL file with a crafted name. This enables another, more severe vulnerability: the table `configuration` stores a `configuration_value` and a `configuration_title` for each setting. Furthermore, a `use_function` can be specified optionally to deploy the configuration's value.

```
1 $sql_file = new upload('sql_file');
2 $read_from = $sql_file->filename;
3 tep_db_query("INSERT INTO " . TABLE_CONFIGURATION .
4    " VALUES (null, 'Last Database Restore', 'DB_RESTORE',
5    '" . $read_from . "', 'Last database restore file',
6    '6', '0', null, now(), '', '')");
```

Listing 4.11: Simplified code of the *backup.php* file in *osCommerce* shows a SQLi through a file name.

When the list of configuration values is loaded from the database, the function name specified in the `use_function` column is called with the `configuration_value` as argument (see Listing 4.12, line 8). An attacker can abuse the SQLi to insert an arbitrary PHP function's name, such as `system`, to the column `use_function` and insert an arbitrary argument, such as *id*, to the column `configuration_value`. When loading the configuration list, the specified function is fetched and called with the specified argument that executes the system command *id*.

```
1 $conf_query = tep_db_query("SELECT configuration_id, configuration_title,
2                                    configuration_value, use_function
3                             FROM " . TABLE_CONFIGURATION . "
4                             WHERE configuration_group_id = '" . (int)$gID . "'");
5 while ($configuration = tep_db_fetch_array($conf_query)) {
6    if (tep_not_null($configuration['use_function'])) {
7       $use_function = $configuration['use_function'];
8       $cfgValue = call_user_func($use_function, $configuration['configuration_value']);
9    }
10 }
```

Listing 4.12: Simplified code of the *configuration.php* file in *osCommerce* demonstrates a *multi-step RCE*.

Sanitization Bypass in osCommerce

Another *multi-step RCE* exploit was reported in *osCommerce* that involves a sanitization bypass. The previously mentioned backup tool of the administrator interface allows to specify a local ZIP file that is unpacked via the system command *unzip*. Here, the target file name is specified as an argument in the command line if the specified file name exists on the file system. The simplified code is shown in Listing 4.13. An attacker can bypass this check by abusing one of the file upload functionalities in *osCommerce*. By uploading

a file with the name *;id;.zip* and then specifying this file as backup file, the command *id* is executed. The semicolons within the file name terminate the previous *unzip* command and introduce a new command.

```
1 if (file_exists(DIR_FS_BACKUP . $HTTP_GET_VARS['file'])) {
2    $restore_file = DIR_FS_BACKUP . $HTTP_GET_VARS['file'];
3    exec(LOCAL_EXE_UNZIP . ' ' . $restore_file . ' -d ' . DIR_FS_BACKUP);
4 }
```

Listing 4.13: A dynamically constructed system command in *osCommerce* includes the name of an existing file.

4.2.4 False Positives

In total, our prototype generated 43 false second-order vulnerability reports, leading to a false discovery rate of 21% for our selected applications. All false positives base upon the fact that our prototype is not able to detect path-sensitive sanitization (see Section 3.10). Thus, false *persistent XSS* vulnerabilities were reported in *Scarf* and *HotCRP* that leverage email addresses from the database. Our prototype erroneously identified these columns as taintable where, in fact, the addresses were sanitized from malicious characters (see Section 4.2.2). The same error applies to a user-supplied paper format in *OpenConf* which led to four false positives. A user-defined sanitization function using path-sensitive sanitization based upon its arguments led to 29 false *persistent XSS* reports in *osCommerce*. Furthermore, a false multi-step exploit was reported in *HotCRP* caused by a false SQLi report. By performing a path-sensitive sanitization analysis, these false positives can be addressed in the future.

4.2.5 False Negatives

Evaluating false negatives is an error-prone task because the actual number of vulnerabilities is unknown. Furthermore, no CVE entries are public regarding second-order vulnerabilities in our selected applications. However, it is possible to test for false negatives that stem from insufficient detection of taintable PDS. By pre-configuring our implementation with the taintable PDS we identified manually, we can compare the amount of detected second-order vulnerabilities with the number of reports when PDS are analyzed automatically.

As a result, only six previously missed *persistent XSS* in *osCommerce* were reported. Additionally, another taintable session key in *OpenConf* was reported, although the key does not lead to a vulnerability. Furthermore, we manually inspected the source code of the applications and observed that our SQL parser needs improvement. Three false negatives occurred in *Scarf* because our parser does not handle all SQL string functions, such as `concat()`. More complex SQL instructions might lead to further false negatives but are used rarely.

4.3 POP Chain Detection

In a third experiment, we added our novel approach for object- and field-sensitive data flow analysis, as well as for the automated POP gadget chain generation to our prototype. We then examined the CVE database regarding *PHP object injection* vulnerabilities in modern PHP applications [88]. Out of the CVE entries published in the years 2013 and 2014, we chose applications according to the following criteria:

- The vulnerable software version is still available for download so that we can replicate the vulnerability.
- The application is non-trivial (i. e., has more than 40 KLOC) and is primarily written in object-oriented code.
- The affected application is exploitable *as it is*. For example, we excluded third-party plugins or framework components that require an implementation.

We selected nine CVE entries matching to our criteria and also added Piwik as the first reported software vulnerable to POI [111]. The list of selected applications is given in Table 4.11. In total, our prototype was able to find 30 new vulnerabilities and 28 previously undocumented chains. Overall, our evaluation results show that 2 out of 13 known POI vulnerabilities and 2 out of 10 known chains were missed by our current prototype implementation. False positives occurred only during the chain detection in one application. Furthermore, our evaluation showed that `__toString()` and `__destruct()`, the third and fourth most frequent magic methods, are the most prevalent initial gadgets for chains.

In the following, we present our performance results for the detection of POI vulnerabilities and the generation of POP chains in object-oriented PHP code (see Section 4.3.1). Then, in Section 4.3.2, we study how many initial gadgets for a potential chain generation are available in our selected applications. Finally, we evaluate the reported PHP object injection vulnerabilities for each application (see Section 4.3.3) and how many gadget chains our prototype was able to connect to a new vulnerability (see Section 4.3.4).

Table 4.11: Evaluation results for selected applications recently affected by a POI vulnerability. The number of POI vulnerabilities and chains detected by our prototype are compared to the number of previously known issues. **Highlighted** numbers indicate the detection of novel POI issues or POP chains.

CVE Number	Software	Version	Files	LOC	Time	Mem	POI	Gadgets	Chains
CVE-2014-2294	OWA	1.5.6	463	82 013	155	475	0/1	24	**9/0**
CVE-2014-1860	Contao CMS	3.2.4	578	202 993	298	1 264	**19/3**	136	**14/3**
CVE-2014-0334	CMSMadeSimple	1.11.9	692	135 478	567	922	1/1	41	**1/0**
CVE-2013-7034	LiveZilla	5.1.2.0	103	42 753	151	342	**2/1**	21	0/0
CVE-2013-4338	Wordpress	3.5.1	425	190 800	1 138	7 640	0/1	41	0/0
CVE-2013-3528	Vanilla Forums	2.0.18.5	597	123 465	951	6 471	2/2	14	0/1
CVE-2013-2225	GLPI	0.83.9	1 025	347 682	676	1 632	**15/1**	77	0/0
CVE-2013-1465	CubeCart	5.2.0	846	141 404	447	1 483	1/1	47	**3/1**
CVE-2013-1453	Joomla	3.0.2	1 592	289 207	338	1 251	**2/1**	73	**5/2**
CVE-2009-4137	Piwik	0.4.5	750	174 314	87	476	1/1	111	**4/3**
Total			7 071	1 730 109	4 808	21 956	**43/13**	585	**36/10**

4.3.1 Performance

As listed in Table 4.11, we subsequently analyzed 10 applications with a total of 1.73 million LOC. The time is denoted in seconds and the memory consumption (*Mem*) is denoted in megabytes. On average, each of our selected application consists of 700 PHP files and about 170 KLOC. Approximately, our prototype implementation required 8 minutes and about 2 GB of memory to perform the POI and POP analysis for a given application. In other words, nearly 13 MB and 2.78 seconds per KLOC were needed. Compared to our evaluation of non-object-oriented code (see Section 4.1.1), the required memory and scan time doubled. However, with object-oriented code, the amount of possible states heavily increases and our automated analysis clearly outperforms manual code analyses in time.

Note that the number of available gadgets does not significantly influence the overall performance. That is, first of all, due to the fact that the code size of magic methods is often rather small. Furthermore, some of them are already included into our regular OOP analysis. Secondly, all further gadgets in a chain are user-defined methods. These are analyzed for POI vulnerabilities by our prototype. Because the analysis results are stored in the methods' summary, they can be reused when building a chain with little effort.

4.3.2 Available Initial Gadgets

First, we let our prototype report all declared, non-empty magic methods in our selected applications to establish a ground truth for gadget chain potential. On average, there are about 59 potential initial gadgets available per application. The different amounts of magic methods are listed in Table 4.12. In our evaluation, the most common magic methods are `__set()` and `__get()` methods. However, since they implement the simple logic for missing getter and setter methods, none of them was exploitable. Among the available gadgets, the `__destruct()` method is also frequently present. It provides the best chance for abusable code because it is context-independent. The context-dependent method `__toString()` is defined often, but is supposed to return a string representation of the object which does not yield a high chance of abusable PHP code. The least frequent magic methods are `callStatic()` and `__invoke()` which we did not find in any of our selected applications. Based on the low number of potential initial gadgets, we

Table 4.12: Initial gadget distribution within our selected applications. **Highlighted** numbers indicate usage in our detected gadget chains.

Software	set	get	toString	destruct	isset	call	wakeup	unset	clone	set_state	callStatic	total
Contao CMS	47	32	**16**	**17**	12	3	4	2	2	0	1	136
Piwik	11	19	**23**	**21**	9	9	**8**	8	3	0	0	111
GLPI	43	23	5	1	1	0	0	0	0	4	0	77
Joomla	4	11	30	**15**	1	4	4	1	3	0	0	73
CubeCart	8	11	4	**18**	1	3	0	1	1	0	0	47
Wordpress	4	6	13	8	5	2	0	2	0	0	1	41
CMS Simple	8	15	7	**3**	2	3	1	0	2	0	0	41
OWA	2	2	3	**15**	2	0	0	0	0	0	0	24
LiveZilla	1	6	4	5	1	3	0	1	0	0	0	21
Vanilla	3	3	4	1	0	3	0	0	0	0	0	14
Total	131	128	109	104	34	30	17	15	11	4	2	585

expected a POI in Vanilla Forums (14 ggadgets), LiveZilla (21 ggadgets), and Open Web Analytics (24 gadgets) to be less likely exploitable compared to, for example, Contao CMS (136 gadgets) or Piwik (111 gadgets).

4.3.3 POI Detection in OOP Code

As a next step, we verified if our prototype detects the POI vulnerabilities described in the CVE entries. We compare the number of reported POI vulnerabilities by our prototype to the number of described vulnerabilities in each CVE in the column *POI* of Table 4.11. For 8 out of 10 vulnerable applications, at least one POI was detected. For four applications, our prototype even found at least one novel POI vulnerabilities that is not included in the CVE. We believe that these vulnerabilities were missed during manual analysis. Our prototype reported no false POI vulnerabilities.

The novel POI vulnerabilities are fixed in the latest LiveZilla 5.2.0.1, Contao CMS 3.2.9 and GLPI 0.84.5 by replacing calls to `unserialize()` with `json_decode()`, or by sanitizing user input. However, the POI in CMS Made Simple was not fixed in the latest release yet because no chain was known. Our prototype detected a novel gadget chain to delete arbitrary files and we reported the issue to the developers. Our novel POI in Joomla also exists in the latest version 3.3.0 and we reported the issue as well (*CVE-2014-7228*).

Table 4.13: The distribution of different vulnerability types in our detected POP gadget chains.

Software	FD	FC	FM	SQLi	LFI	XXE
Open Web Analytics	2	2	1	3	1	-
Contao	6	5	3	-	-	-
CMS Made Simple	1	-	-	-	-	-
CubeCart	1	-	-	2	-	-
Joomla	1	-	2	1	1	-
Piwik	-	2	1	-	-	1
Total	11	9	7	6	2	1

4.3.4 Detected POP Gadget Chains

Next, we evaluated the reported POP gadget chains of our prototype. For Wordpress and Open Web Analytics, we simplified the POI vulnerability so that our prototype was capable of detecting the vulnerability after which we can include the applications in our gadget chain evaluation.

The total number of exploitable gadget chains reported by our prototype is compared to the known gadget chains from security advisories in the column *Chains* in Table 4.11. In total, 36 exploitable gadget chains were reported. Our prototype successfully detected a gadget chain in 6 out of 10 applications, whereas 28 gadget chains were previously unknown. Starting from the initial gadget to the sensitive sink, the length of detected gadget chains ranges from 1 up to 8 gadgets with an average chain length of 3 gadgets. Table 4.12 highlights the magic methods used as an *initial gadget* with a bold number. The

most abused magic method was `__destruct()`, used by 86% of the gadget chains. Only four gadget chains initially exploited `__toString()`, and one chain exploited `__wakeup()`.

The number of different vulnerability types detected in each application through POP is listed in Table 4.13. The most prominent vulnerability types are *file delete* (FD), *file create* (FC), and *file modification* (FM) vulnerabilities. Furthermore, SQLi and LFI vulnerabilities were detected, as well as one *XML eXternal entity* injection (XXE). We published an advisory describing the technical details of the chains reported in Joomla [29].

Surprisingly, 9 chains were found in Open Web Analytics, although only 24 initial gadgets are available. However, one call to a method with a frequently used name is enough to jump to a large portion of the application's code. Due to one dynamic class invocation (refer to Section 4.3.3) also 10 false positives occurred. For LiveZilla, Wordpress, and GLPI, no gadget chain was detected by our tool. However, since no gadget chain is publicly documented, we assume that the POI vulnerability is not exploitable with the application's core code.

4.3.5 False Negatives

The POI vulnerability in Open Web Analytics and Wordpress was *not* detected by our prototype. The root cause for the false negative in Open Web Analytics is the insufficient analysis of reflection, which is an unsolved problem in the field of static analysis [16,55,82].

```
1 class owa_coreAPI {
2    public static function classFactory($module, $class) {
3        return owa_lib::factory(OWA_BASE_DIR.'/modules/'.$module.'/classes/', $class);
4    }
5    public static function getRequestParam($name) {
6        $service = owa_coreAPI::classFactory('base', 'service');
7        return $service->request->getParam($name);
8    }
```

Listing 4.14: Dynamic class factory in Open Web Analytics.

The simplified code is shown in Listing 4.14. In Open Web Analytics, every access to user input is performed via the static method `getRequestParam()` which is defined in line 5. This method fetches a new object through the method `classFactory()` in line 6 and calls the method `getParam()` on the `request` property as receiver. Because the method `factory()` used in `classFactory()` internally uses reflection, no knowledge about the object assigned to `$service` in line 6 is available to our prototype. The prototype can still fingerprint the method `getParam()`, but this method accesses properties of the object assigned to the property `request`. Its properties are filled during the dynamic object construction in the factory and hence are invisible to our approach. Similar false negatives based on complex object-oriented code occurred for gadget chains in Piwik and in Vanilla Forums. We plan to improve the analysis of dynamic OOP code in the future.

The false negative in Wordpress is based on second-order data flow [31]: metadata about a user is stored in a database and later loaded into a cache before it is deserialized. The database queries are constructed dynamically and cannot be reconstructed completely by our prototype in order to recognize the data flow.

4.4 Discussion

In this chapter, we evaluated our prototype implementation in three experiments built upon each other. First, we evaluated the precision and performance of our refined taint analysis for five popular PHP applications. Our evaluation showed that current taint-style vulnerabilities base upon complex PHP features which are not detected by existing tools and that our prototype outperforms state-of-the-art tools designed for the detection of traditional vulnerability types. Next, we evaluated our novel approach for the detection of second-order vulnerabilities with six well-known PHP applications. We demonstrated that our prototype successfully detects critical second-order vulnerabilities and multi-step exploits which are not supported by prior work. In a third experiment, we evaluated our novel algorithms for the analysis of object-oriented code. We showed that our prototype is the first to automatically generate POP gadget chains and that it is able to analyze modern PHP applications with over 200 KLOC in under 10 minutes. By optimizing the prototype implementation of our algorithms, the performance can be further improved.

In total, our prototype detected 321 previously unknown vulnerabilities, including vulnerabilities in some of the most-popular PHP applications on the Web [146] with a total false discovery rate of 20 %. Contrarily, other approaches proposed in previous work evaluated only less popular applications with a fraction of code sizes [4, 66, 74, 121, 154]. In our evaluation of those applications for taint-style and second-order vulnerabilities, the false discovery rate dropped down to 2 %. Excluding these applications, we detected in *modern* applications 73 TP and 29 FP taint-style, 114 TP and 34 FP second-order, as well as 58 TP and 10 FP POP chain vulnerabilities. This leaves us with a false discovery rate of 23 % for modern PHP applications (i. e., `73 FP/(73 FP+245 TP)`) and, on the contrary, a precision of 77 %. The main root cause for false positives in our prototype implementation is the path-insensitivity of our taint analysis [14, 33, 36, 159].

Moreover, the detection of 51 out of 65 known vulnerabilities showed that our prototype has an estimated recall of 78 %. However, it remains unknown how many vulnerabilities are truely hidden in our selected applications. The main reason for false negatives in our evaluation were dynamic control flows, such as *reflections*. Both issues can be addressed in the future but will likely affect the scalability, so that a new tradeoff between precision and performance will be required. Note that our evaluation includes a set of 23 applications and a different corpus could lead to anomalous precision and recall rates. Although the good performance of our prototype allows to analyze a multitude of the selected applications, the evaluation of all reported issues is time-intense.

Last but not least, the benefit of a static code analysis tool is heavily influenced by its output, usability, customizability, result understandability, and its workflow integration. These factors were not included into our evaluation but play an important role for the tool's adoption [64, 117]. In fact, the best precision is rendered worthless if the user spuriously flags true positives as false positives because of incomprehensibility. Hence, a taint analysis algorithm must also be able to track and store all relevant information about a security vulnerability for the users understanding. This poses further challenges to an efficient algorithm implementation which are out of scope for this work. Nonetheless, by observing our prototype's performance in large-scale studies, as presented in the next chapter, we can optimize our algoritms and our analysis results.

Chapter 5

Empirical Studies

As demonstrated in the last chapter, our prototype is able to precisely analyze real-world PHP code for security vulnerabilities. In this chapter, we instrument our tool to empirically study the nature of security vulnerabilities from a different perspective. Namely, we have a look at how *developers* attempt to secure their applications from vulnerabilities and, also, which actions *attackers* perform after an exploitation. First, in Section 5.1, we introduce related studies. Then we present our two empirical studies, including the necessary prototype modifications, the results, and our lessons learned.

In Section 5.2, we provide an empirical analysis of 20 different data sanitization and validation mechanisms which we found being used in web applications. Furthermore, we study common implementation pitfalls that lead to vulnerabilities in practice. We extended our prototype for enumeration and applied it to 25 of the most popular web applications. We analyzed more than 2.5 million lines of code and present an analysis of the detected security patterns in more than 26 thousand data flows. Our analysis helps to answer the following essential questions for PHP developers, code auditors, and static analysis engineers in order to refine the focus during vulnerability detection:

- **Q1**: Which security mechanisms are used how often in modern PHP applications?
- **Q2**: Which security mechanism is used to prevent which vulnerability type in which markup context?
- **Q3**: Which pitfalls occur in practice?

In Section 5.3, we perform a comprehensive study on popular PHP shells, scripts uploaded by attackers to compromised web servers. We utilize the vulnerability detection of our prototype in order to discover and quantify the most frequently provided features in a unique set of almost 500 shells. Furthermore, we manually audited the shells' authentication mechanisms for backdoors. Our study allows to answer the following questions about the nature of PHP shells and the surrounding ecosystem:

- **Q4**: Which features in popular PHP shells are commonly available to attackers?
- **Q5**: How many popular PHP shells are backdoored?

5.1 Related Work

We now first discuss related work that is close to our studies. Next to security vulnerability trends [23, 39, 145], empirical studies regarding sanitization and validation approaches in PHP applications were conducted. However, the covered mechanisms are either incomplete or studied in a different context. Web shells have been only treated as malicious "black boxes" so far that need to be detected [141], rather than understood.

5.1.1 Security Mechanisms

Hills et al. conducted an interesting study of the usage of PHP features in 19 applications [55]. Among different features, the occurrences of typecasts and binary operations were studied. However, these features were not interpreted regarding security and no other security mechanisms were covered.

Scholte et al. empirically analyzed the data type of the parameters that are affected by XSS and SQL injection for over 7 000 vulnerabilities [108]. Next to native data types, such as string, integer, and boolean, they also considered custom data types, such as *email*, *url*, or *username*. As a result, the most commonly affected data types were identified, as well as the lack of built-in sanitization mechanisms for these types in common web frameworks.

Weinberger et al. empirically studied present sanitization approaches against XSS in web application frameworks [152]. They analyzed the availability of sanitization approaches for different HTML markup contexts for five PHP frameworks. Furthermore, eight PHP applications were studied for the usage of various markup contexts.

In his dissertation about decision procedures for string constraints, Hooimeijer studied the occurrence of 113 PHP built-in string functions in 88 applications [57]. Among these functions are length limiting, regular expression, and formatting functions, but these are not interpreted regarding sanitization or validation.

Saxena et al. developed *ScriptGard* to detect and correct the misplacement of sanitizers in ASP.NET applications with dynamic analysis [107]. They studied one application with 400 KLOC for context-mismatched sanitization or sanitizer sequences. In comparison, we studied 25 applications and 2.5 million LOC.

5.1.2 Web Shells

Canali and Balzarotti, in their large-scale web honeypot experiment, deployed 17 publicly accessible web shells [20]. The authors noticed that if attackers discover a vulnerability that allows them to upload files, in 46% of the cases, they will upload a web shell and use that shell to interact with the compromised server.

In a follow up study, Canali et al. [21] uploaded the c99 shell (arguably the most popular shell) to 22 shared hosting providers. As a result, only one of the 22 investigated shared hosting providers identified the malicious shell, even among the providers who were offering security services at an additional cost.

Kim et al. [72] benchmarked the detection rate of three popular shell-detecting tools. The authors discovered that these tools either detected only well known shells and ignored less known ones, or marked a large number of benign files as "suspicious", offloading the work of verifying a script's maliciousness to a human analyst.

5.2 Security Mechanism Usage

Although some best-practice guidelines on secure web programming emerged (e. g., recommendations by OWASP [97]), no comprehensive security standards are available for developers. This leads to the observation that each developer applies his own favorite security mechanisms for data sanitization or validation. As a result, many programming patterns emerged for input sanitization (e. g., type casting, data encoding, converting data to HTML entities, or prepared statements) and input validation (e. g., type validation, format validation, or whitelisting). As we introduced in Section 2.3, each of these patterns has its own advantages and drawbacks, and programming mistakes due to common pitfalls can still lead to vulnerabilities.

Based on our extended prototype, we empirically study the usage of different security mechanisms in combination with the markup context. This allows us to evaluate common and uncommon combinations, as well as, associated pitfalls. We first introduce the software picked (see Section 5.2.1) and our methodology of enumeration (see Section 5.2.2). Then we present our results in Section 5.2.3 that provides answers to research question **Q1**–**Q3** in detail and discuss our lessons learned in Section 5.2.4. Threats to validity of our study are addressed in Section 5.2.5.

Table 5.1: Overview of 25 selected applications with the amount of analyzed lines of code (LOC) and detected markup injections in HTML, SQL, and JavaScript (JS).

Software	Version	LOC	HTML	SQL	JS
Beehive	1.4.3	105 325	2 976	402	0
CMSSimple	1.11.11	137 222	190	335	1
Concrete5	5.6.3.1	317 025	1 823	161	109
Couch CMS	1.4	37 073	25	29	16
e107	1.0.4	157 706	2 561	828	937
FluxBB	1.5.6	28 945	268	145	2
FreePBX	2.11.0.25	75 909	147	36	10
FUDForum	3.0.6-RC2	74 421	556	211	31
HotCRP	2.92	40 865	181	106	3
LiveZilla	5.2.0.1	43 593	40	181	0
Nucleus CMS	3.6.5	38 268	61	49	1
OpenConf	6.0	21 836	325	180	2
osCommerce	2.3.4	85 563	2 615	788	0
Phorum	5.2.19	73 841	304	699	2
PHP Fusion	7.02.07	54 584	805	563	40
PHP Nuke	8.3.2	200 767	261	291	0
phpList	3.0.6	103 647	670	169	15
Pligg CMS	2.0.1	62 588	24	258	0
PunBB	1.4.2	43 268	119	84	2
Roundcube	1.0.2	158 435	179	24	0
Serendipity	1.7.8	212 705	137	418	19
Squirrelmail	1.4.22	56 194	1 097	123	3
Wacko Wiki	5.4.0	103 217	100	132	0
Xoops	2.5.6	142 749	209	74	10
Zen Cart	1.5.1	131 458	1 029	1 810	5
Sum		2 507 204	16 702	8 096	1 208
Average		100 288	668	324	48

5.2.1 Corpus

In order to obtain the most precise results, we carefully chose the applications for our study. First, we gathered a coarse list of PHP applications according to the following three criteria:

- The application is open source, active, and popular according to W3Tech's usage statistic [146].
- The application has an size of at least 20 KLOC.
- The application works standalone and does not require additional code.

Then, we excluded applications from our list that make an extensive use of reflection or application frameworks. As mentioned in Section 3.10, static analysis of these components is limited and often requires manual configuration [122]. We discuss related threats to validity in Section 5.2.5. The list of our 25 selected applications for evaluation is given in Table 5.1.

5.2.2 Methodology

An erroneous approach to count security mechanisms in an application would be to count the occurrences of security related operators and built-in functions in the code. On the one hand, this leads to an over-approximation when these features are used for other purposes than for data sanitization or validation, such as a type-cast of non-sensitive data. On the other hand, this leads to an under-approximation when these features are declared in reusable code once but called multiple times at runtime, such as a user-defined function.

A more precise enumeration of security mechanisms is achieved by leveraging static data flow and taint analysis. Here, the security relevance of data flow through such a mechanism can be evaluated by the taint status of the data. A mechanism should only be associated with security if it sanitizes or validates *tainted* data and this data reaches a sensitive sink. For this purpose, we use the data flow and taint analysis of our prototype and leverage its *sanitization tags* (see Section 3.7). Furthermore, we extended our prototype by adding *pitfall tags*. These are inferred from the AST during block simulation (see Section 3.4) for delusive input sanitization, such as the increment operator (see Section 2.3.1), or weak validation, such as fragmentary regular expressions (see Section 2.3.4). Similarly, we add a pitfall tag for delusive input validation to BOOLEAN data symbols during our edge simulation (see Section 3.5).

Next, we extended the taint analysis of our prototype in order to log a successfully applied sanitization mechanism that prevents a vulnerability. For this purpose, we added the mechanism's name to the applied sanitization tag of a data symbol. Furthermore, we log a taken pitfall if a tainted data symbol is used in a sensitive sink and owns invalid sanitization tags, insecure encoding, or dispensable escaping regarding the detected markup context. Additionally, a pitfall is logged if no correct sanitization tags are found but a *pitfall tag* was assigned to the tainted data symbol.

We limit our study to XSS and SQLi vulnerabilities because these are the most prevalent injection flaws. Both types have a variety of contexts by using a separate markup language (namely HTML and SQL). A security mechanism is counted whenever a *unique* source flows into a *unique* markup context of a sink and was sanitized or validated correctly by a security mechanism. Incorrect sanitization or validation regarding the markup context is counted as pitfall, as well as any present pitfall tag. Our backwards-directed taint analysis allows us to count only the nearest security mechanism before the sink. In the case of a path-sensitive sanitization or termination, the corresponding *validation* mechanism is counted (refer to Section 2.3.5). The detection of path-sensitive validation is disabled (reasons in Section 3.10). Furthermore, we excluded second-order tainting [31] and validation (see Section 2.3.3) because it depends on the analysis of *taintable* resources which is more error-prone than the detection of other security mechanisms.

5.2.3 Results

In total, we analyzed 2.5 million lines of code and 26 006 unique data flows where a source *flows* sanitized (53 %) or validated (47 %) into a sensitive sink. Data flows without any applied security mechanism are excluded from our study.

As shown in Table 5.1, the most common markup context we found is HTML (64 %), followed by SQL (31 %) and JavaScript (5 %). This is likely related to the fact that an average application prints more data to the response page than that it interacts with the database [55]. The CSS context, for example within a `style` attribute or tag, appeared rarely and was excluded from our study.

As a preliminary answer to research question **Q1**, we found that user input is primarily secured with a type validation (19 %) or an explicit typecast (16 %). The extensive use of type-related security mechanisms shows the additional work on the developer side that would otherwise be handled by default in a strongly typed language. Other security mechanisms are applied context-sensitively to the markup and are revealed throughout this section in order to answer research question **Q2**. Format validation or PHP's valuable filter functions are the least detected security mechanisms in our test set (<1 %). Instead, string replacement and regular expressions are used, which are the security mechanisms with the highest pitfall density in our test corpus (**Q3**). In the following, we examine our results for the HTML, JavaScript, and SQL markup in detail.

HTML Markup Security

Table 5.2 shows the seven most frequent security mechanisms used to secure the five most common HTML markup contexts, according to our study. For example, 3 367 occasions were detected where user input within a double-quoted (*DQ*) HTML attribute was correctly sanitized by converting characters.

Table 5.3 shows frequent markup contexts where this mechanism failed, for example for 3 URL attributes. In summary, the HTML landscape is dominated by double-quoted HTML attributes (52 %) and the context between two HTML elements (41 %), followed by single-quoted (*SQ*) attributes context (5 %). HTML comments, fully controlled URL attributes, and attribute names are rare contexts (<1 %).

Table 5.2: Mechanisms safely applied to HTML contexts.

	DQ Attr	Element	SQ Attr	Comment	URL Attr
Converting	3 367	839	18	8	10
Type Validation	1 816	1 494	15	27	0
Comparing	999	1 269	32	35	95
Explicit Typecast	439	1 075	34	57	20
Regex Validate	903	397	145	1	1
String Replace	27	513	517	3	1
Null Validation	167	205	9	15	0
Other	918	869	48	37	16
Sum	8 636	6 661	818	183	143

Table 5.3: Pitfalls triggered in HTML contexts.

	Element	DQ Attr	SQ Attr	URL Attr	Comment
String Replace	57	12	3	2	1
Regex Validation	33	10	6	0	0
Escaping	40	0	0	0	0
Regex Replace	20	11	4	0	2
Comparing	7	18	8	1	0
Converting	0	0	6	3	0
String Search	4	2	0	0	0
Other	2	9	0	0	0
Sum	163	62	27	6	3

The most frequently used security mechanism is an HTML character conversion by PHP built-in functions, such as `htmlentities()` (see Table 5.2 and Table 5.3, *cursive*). This data sanitization is applied to 25 % of all 16 702 detected HTML markup contexts, primarily to the two markup contexts it is designed for: double-quoted attributes and between HTML elements.

However, we also detected 24 cases where it is applied to a single-quoted attribute context that requires an additional parameter to the `htmlentities()` function (see Section 2.3.2). In 6 cases (e. g., in FreePBX), this pitfall was triggered which leads to the suggestion to always use double-quoted HTML attributes. Furthermore, type validation (20 %) and explicit typecasting (10 %) is regularly applied, followed by data validation with regular expressions (9 %) and data sanitization with string replacements (7 %).

In particular, the two latter are prone to pitfalls, as shown **highlighted** in Table 5.3. We found that 7 % of all applied string replacements and 3 % of all applied regular expressions to tainted data are insufficiently sanitizing or validating the HTML context. A single forgotten character that slips through the string replacement or regular expression filter leads to an exploitable vulnerability.

Moreover, cross-site scripting occurs when data is sanitized for a *different* vulnerability type. For example, in 40 occasions data is correctly escaped for the use in a SQL query, but later printed insecurely to the HTML markup between two elements.

Table 5.4: Correctly secured JS context.

	Script tag	Eventhandler
Type Validation	465	2
Explicit Typecast	206	27
String Compare	161	7
String Replace	80	2
Null Validation	31	4
Regex Replacing	31	2
Encoding	26	7
Other	75	17
Sum	1 075	68

Table 5.5: Triggered pitfall in JS context.

	Script tag	Eventhandler
Converting	34	8
String Replace	14	0
String Compare	4	0
Regex Validation	3	0
Regex Replacing	1	0
String Truncation	1	0
Decoding	0	0
Other	0	0
Sum	57	8

JavaScript Markup Security

As shown in Table 5.4, the prevalent JavaScript context in our selected applications is the HTML `script` tag that clearly dominates over eventhandler attributes. The table lists the top seven security mechanisms applied to both contexts, as detected by our prototype in the corpus. Next to type-related security mechanisms, such as type validation (39 %) and explicit typecasts (19 %), many custom sanitization approaches are found that base on string comparison (14 %), regular expressions (3 %), or string replacement (8 %).

Similar to the HTML markup, the latter is the root cause for 14 pitfalls (see Table 5.5, **highlighted**). For example, in Couch CMS the backslash and double-quote character is replaced to prevent an outbreak of double-quotes. However, an attacker can terminate the current `script` tag and start a new JavaScript context that requires no quotes by injecting `</script><script>`. According to our study, the most commonly taken pitfall for a JavaScript context is based on character conversion. In fact, we found more vulnerable applications of character conversion to a JavaScript context than secure ones. This is related to the fact that most JavaScript contexts use no quotes or single-quotes, which are not converted by the built-in functions by default.

SQL Markup Security

Table 5.6 and Table 5.7 compare the distribution of correctly and wrongly applied security mechanisms to 8 096 detected SQL contexts. In our evaluation, 67 % of the sources are embedded into single-quotes (SQ), 31 % without quotes (NQ), and 2 % into double-quotes (DQ). The values are usually sanitized (27 %) or validated (14 %) by data type, or sanitized by escaping (19 %). However, for 72 escapes (4 %), the surrounding quotes are forgotten which leads to SQL injection. This is the most frequent pitfall encountered for SQL markup. Contrarily, we found that only 5 % of all SQL queries in our selected applications use prepared statements which would prevent these obstacles. Moreover, 7 % of all these prepared statements are handled unsafe (e. g. in PunBB), indicating that the concept is not thoroughly understood. As for the HTML and JavaScript markup, many vulnerabilities stem from insufficient string replacement or regular expression validation (see Table 5.7, **highlighted**). While these mostly recognize quotes within the input, the SQL markup misses quotes or the backslash character is forgotten.

Table 5.6: Secured SQL context.

	SQ	NQ	DQ
Explicit Typecast	1 140	1 028	1
Escaping	1 519	0	38
Type Validation	826	287	6
String Compare	537	248	19
Regex Validation	497	77	4
String Replace	382	39	4
Prepared Statements	0	352	0
Other	489	316	49
Sum	5 390	2 347	121

Table 5.7: Pitfalls in SQL context.

	NQ	SQ	DQ
Escaping	72	0	0
Regex Validation	26	11	0
String Replace	18	15	2
String Compare	15	12	0
String Truncation	7	20	0
Prepared Statements	21	2	0
Regex Replacing	9	1	0
Other	2	4	0
Sum	170	65	2

5.2.4 Lessons Learned

Based on our results, we were able to suggest answers to our research questions regarding the diversity of security mechanisms and pitfalls (**Q1**-**Q3**). This provides valuable insight for the practice and teaches the following lessons.

First of all, we learned about pitfall-prone markup context. In order to find vulnerable code as a developer, code auditor, or static analysis engineer, an increased focus can be applied to markup contexts with a high *pitfall density* (detected pitfalls per detected markup contexts) and high frequency (see Figure 5.1). According to our evaluation, fully controlled URL attributes (18 %) and eventhandlers (15 %) are highly prone to pitfalls. However, these appear seldom in code (0.2 % of all detected markup contexts). More commonly in practice are SQL values with no quotes (9 %), `script` tags (4 %), and single-quoted HTML attributes (3 %) that have a pitfall density of 8 %, 6 %, and 4 %, respectively (according to our analysis results). These are the contexts to keep an eye on. Least likely affected by pitfalls are double-quoted HTML attributes and single-quoted SQL values (1 %).

Second, we learned about pitfall-prone security mechanisms. As **highlighted** throughout Table 5.2-5.7, custom security mechanisms based on regular expressions and string replacement are more prone to pitfalls than other security mechanisms, but appear rather frequently. These should be carefully inspected, and often can be replaced with a less error-prone security mechanism. We also believe that a high pitfall density for JavaScript markup is related to the fact that there is no designated built-in function in PHP.

Last but not least, we use the lessons learned as a new metric for our tool to improve the severity ranking of vulnerability reports. Except for the *universal* sanitization tag that introduces performance issues (see Section 3.10), we kept the logging of applied security mechanisms in our prototype. This allows us to rank vulnerabilities detected in error-prone markup contexts, such as an XSS within an URL attribute, higher than reports in other markups that are statistically less vulnerable. Similarly, reported vulnerabilities with a pitfall in string replacement or a regular expression are ranked higher. The logging also helps us to validate our true negative rate and to detect a low code coverage. We assume that a large application will result in a certain amount of logged security mechanisms and markup contexts. Otherwise, this indicates a low code coverage that could possibly stem from a template engine or a SQL builder.

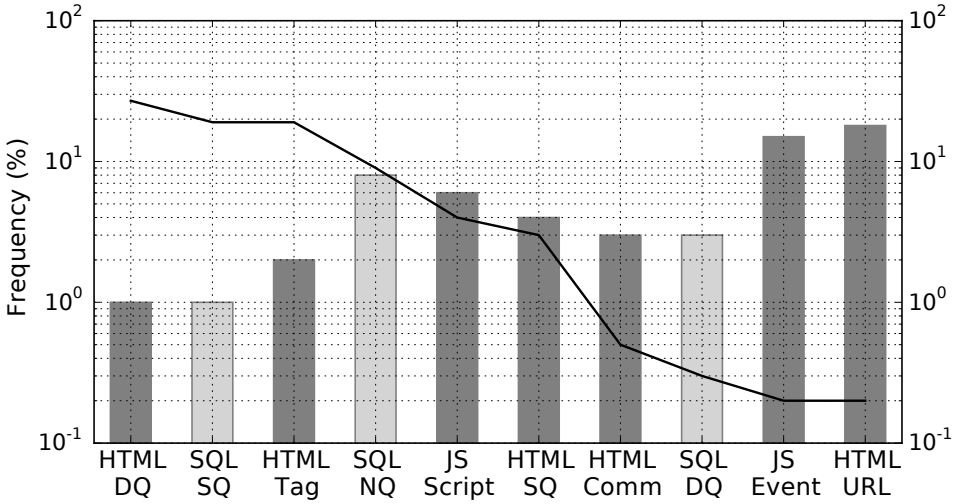

Figure 5.1: Pitfall density (bars) versus frequency (line) of markup contexts prone to XSS (dark) and SQLi (light).

5.2.5 Threats to Validity

There are certain threats to validity of our results that caution to draw strong conclusions and to generalize. Our corpus consists of only 25 popular applications. We excluded some popular applications, such as Wordpress, Joomla, and Drupal, because for these highly dynamic applications, a low code coverage of our static analysis tool is expected, which could tamper the results. We refrained from adding more unpopular applications because we believe that they are more vulnerable. Although more studies need to be conducted to verify our results for more applications, we believe that our corpus represents a comprehensive set of modern and popular applications in order to provide reasonable indicators. However, a different corpus may introduce a different amount of security mechanisms and pitfalls and the detection rate of our tool may be different for each application.

Moreover, as discussed in Section 3.10, static code analysis is limited. Although we would like to find all vulnerabilities present in an application, static analysis allows only to detect a fraction. While we experienced good results with our prototype in the past and verified random samples, we cannot guarantee the absence of false positives, false negatives, or mistakes. We tried to mitigate this threat by excluding applications not suited for our prototype and the results appear to be reasonable according to our experience with security vulnerabilities in PHP applications [30–32].

Finally, our analysis cannot reason about the intention of the developer. This poses a threat to the evaluation of security mechanisms and pitfalls because, for example in case of a string comparison, data can be accidentally validated or a pitfall might be triggered although no validation was intended in first place.

5.3 Web Shell Features

A *web shell* is a piece of software that attackers upload to a compromised web server in order to maintain a permanent and stealth access. Web shells allow adversaries to navigate and control the compromised server in a convenient way by providing remote access to critical functions, e. g., execution of system commands and upload of arbitrary files. As such, they play a crucial role in modern attacks [56]. Despite their high prevalence in practice and heavy involvement in security breaches, web shells have never been the direct subject of any study. In contrast, web shells have been treated as malicious blackboxes that need to be detected and removed, rather than malicious pieces of software that need to be analyzed and, in detail, understood.

In this section, we address this research gap and report on the first comprehensive study of web shells, focusing on the ones written in the PHP language. We collected a corpus of more than 1 400 shells that we use as the starting point for our analysis. By leveraging different data preparation steps, we remove non-shells and files with shallow differences (see Section 5.3.1). To this end, we are left with almost 500 shells. In a first step, we extend our prototype in order to enumerate functionalities (see Section 5.3.2). As a result, we discover the most popular features and provide an answer to our research question **Q4** in Section 5.3.3. In a second analysis step, we manually analyze the *invisible* features provided by the shells. More specifically, we analyze authentication mechanisms and investigate whether some kind of backdoor or bypass is hidden. The result and answer to research question **Q5** is presented in Section 5.3.4. At the end of this section, we discuss threats to validity of our web shell study (see Section 5.3.5).

5.3.1 Corpus

There are hundreds of different shells on the Web, due to both the presence of multiple versions of the same shell as well as shells that have been used as a base for a new shell. Even though there have been several attempts to collect lists of common web shells [19, 28, 130, 140], none of those sources can provide any guarantees regarding completeness and quality. In this section, we thus describe our process for compiling the set of shells that we used for our experiments.

We started by collecting all shells that we could find in underground hacking websites as well as shells that researchers have observed in their honeypots [19, 28, 130, 140]. We avoided shells that are designed for legitimate administration of servers since these are outside the scope of our study. By combining all the aforementioned sources, we obtained our starting set of 1 449 shells. Next, we assessed and improved the quality of our working set, filtering-out non-shells (such as files containing only JavaScript code), shells written in languages other than PHP, and shells with shallow differences to avoid potential duplicates. We processed our 1,449 shells as follows:

- *Filtering* was done by checking for PHP tags and a file size exceeding an empirically-determined threshold. Through this process, we excluded 53 files and obtained 1 396 potential PHP shells.

- *Normalization* was performed in order to remove non-obvious duplicates, i. e., shells whose cryptographical hashes may be different, yet are near-identical from a syntactic point of view. Our shell normalization process involved the removal of comments, new lines, whitespaces, and semicolons. In addition, we replaced all variable names and function names with a single name. Note that this process is not meant to preserve the correctness of the code. We only use it to cluster shells together and pick the first member of each cluster for further analysis. We empirically found that this simple approach yields sufficient good results to, in practice, detect small differences in different files. Note that more elaborated mechanisms based on an analysis of the *abstract syntax tree*, *fuzzy hashing* [76], or via determining the semantic equivalence [1] could be used in the future, but we found our approach to be sufficient in practice. Using this process, we obtained a set of 804 unique shells.

- *Deobfuscation* was necessary since identical shells identified in the previous step may still be using different obfuscation methods. Hence, we used the state-of-the-art *UnPHP* deobfuscation service [143] to automatically deobfuscated our set of 804 unique shells. UnPHP returned the deobfuscated code of 661 shells (the service was consistently timing out when trying to deobfuscate the remaining 143 shells), which we normalized once more to arrive at a set of 607 unique shells.

- *Manual repairment* was finally necessary for shells which were broken by UnPHP's deobfuscation. This was either because UnPHP was not able to correctly escape all special characters in the deobfuscated statements, or because entire pieces of code were missing from the deobfuscated result. We notified the vendor about these issues. We only used the shells that were either fully syntactically correct or the ones which we could repair with a reasonable amount of manual effort.

We also removed shells which we knew would not provide any interesting results, such as shells that were just printing local system information or were merely sending an email to a predefined email address. At the end of this prefiltering process for static analysis, we arrived at 481 unique shells. Our final data set represents different shell families, including variants of *c99*, *r57*, *WSO*, *B374k*, *NST*, *NCC*, and *Crystal*. However, many variants evolved from a family sibling to a new shell family by applying different code obfuscation techniques, adding new features, or copying code from other shell families. For example, the *c99* shell was extended by a privilege escalation feature and renamed to *c999* shell, while a fraction of the *c99* shell's code can be also found in the *Fx29* shell family. PHP shells that are not explicitly labeled in the source code are thus hard to classify and would lead to inaccurate results.

5.3.2 Methodology

In order to better understand the features that web shells make available to attackers, we analyzed our set of shells with static taint analysis techniques. For this purpose, we used and extended our prototype that is able to detect security vulnerabilities in PHP applications, such as *remote command execution*, *remote code execution*, *mail header injection*, and *SQL injection* vulnerabilities. Furthermore, our tool supports the analysis of

file-related vulnerabilities, such as *file upload*, *file write/create*, *file disclosure*, *permission manipulation*, or *file inclusion* vulnerabilities. Each of these vulnerabilities is interpreted as a feature when detected in the deobfuscated PHP shell's code by our prototype.

Our static analysis tool also performs context-sensitive markup analysis (described in Section 3.7.2). This allows us to automatically inspect the markup that reaches a sensitive sink and to enumerate further features it cannot detect out of the box. For example, when our tool analyzes sensitive sinks that execute system commands, the scanner reconstructs all possible strings (i. e., commands) that reach this sink. At this point we applied regular expressions to identify commands that we found commonly related to a *file dropper* (`wget, curl, lynx, get, fetch`), a *reverse* or *back connect shell* (`perl, python, gcc, chmod, nohup, nc`), or *information gathering* (`uname, id, ver, sysctl, whoami, $OSTYPE, pwd`). The results were complemented by additional feature detection algorithms that we added to our prototype. For example, when a download of a remote file through PHP's built-in file features is requested, a *file dropper* feature is reported. Likewise, when the output of PHP's built-in functions or reserved constants related to system information (e. g., `php_uname()`) is encountered during taint analysis of a sensitive sink that prints data to the HTML response page, a *system information gathering* feature is detected.

Moreover, we used the annotations regarding loops in the control flow graph representation of a given shell in order to classify certain vulnerabilities as features. For example, a *mail header injection* within a loop is interpreted as a *spam* feature and a login attempt to an FTP server or an HTTP basic authentication within a loop is interpreted as a *brute-force* feature. Similarly, when a built-in function is used within a loop to establish a socket connection and the port is the subject of iteration, a *portscan* feature is logged.

5.3.3 Features

The final results of our feature enumeration is shown in Figure 5.2. The most prominent feature, appearing in 69 % of all shells in our collection, is the gathering of system information. More specifically, the current working directory, operating system, and the PHP version is of interest to the attacker. Next, the interaction with the file system is supported by 67 % of the analyzed shells. We summarized detected vulnerabilities regarding file read (54 %), create (54 %), list (40 %), delete (38 %), edit (29 %), and modification of permissions (22 %) to one feature named *file browser*. Separately, we found a file upload feature in 54 % of the analyzed PHP shells. Next to a file browser, we detected a slightly less popular *SQL browser* to list all databases and tables in about half of our shells, as well as rarely available *FTP browsers* in only 13 of our 481 analyzed shells.

The traditional *command execution* feature was detected in 59 % of our shells. Next to arbitrary OS command execution, the shells often provide a prefixed list of commands in the web interface, i. e., the *c99* shell proposes commands to find configuration and password files, or writable directories. Less frequently offered (43 %) is the ability to execute arbitrary PHP code through `eval()` and similar operators. This feature appears in smaller shells that focus on stealthiness rather than feature completeness. Additionally, we found 68 shells that offer arbitrary code execution through *remote file inclusion*. In order to bypass restrictive firewalls, remote command execution can also be bound to a separate port opened by an external program (37 %), preferably written in *Perl* or *C*.

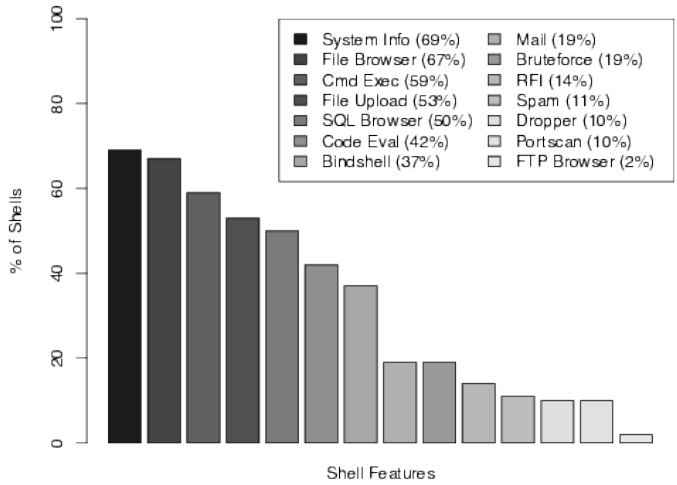

Figure 5.2: Feature distribution in 841 analyzed shells.

Other features are less frequently available and therefore likely less important for an attacker as a core feature of a PHP shell. For example, only 19 % of the shells allow to send an email and 12 % of the shells allow to send out multiple emails (which likely represents spam). The feature to launch bruteforce attacks against FTP and HTTP authentication credentials (19 %) or port scans (10 %) are also rarely available in practice. Note that these tasks can be performed independently from the attacked server. However, when used as part of the PHP shell, the attacker is able to hide her IP address while the compromised web server acts as a proxy.

5.3.4 Authentication Bypasses

We manually examined our set of 481 deobfuscated shells for authentication mechanisms that restrict the shell access to users that possess a secret (e. g., a password). At the same time, we audited these mechanisms for possible bypasses. Our intuition is that malicious attackers do not freely publish their shells as a service to "fellow" attackers, but rather to gain access to servers that are compromised by uncautious attackers using that shell.

Most of the authentication mechanisms found were simply based on username/password credentials, either supplied by HTTP basic authentication or an HTTP POST request via a HTML login form. Furthermore, access was limited to a given IP address or IP address range. We also observed samples that require a secret key supplied via hidden HTTP GET parameter or user agent, as well as samples that expect a password which is then used as a XOR key to decrypt the evaluated PHP code. Most of the detected mechanisms, however, default to no required authentication, for example, if the password in the configuration code is left empty. Listing 5.1 shows an authentication mechanism that is used in 23 of our shells. Here, the variable `$md5` is left empty and the session key *login* is automatically set to `true` which authenticates the user's session. If the variable `$md5` is initialized with the MD5 hash of a password, however, the user has to provide the correct password for authentication via an HTTP POST request before he can access the shell's features. Thus, by setting a password in the shell's code, the authentication mechanism is activated.

```
1  session_start();
2  // configuration
3  $md5 = '';
4  // authentication check
5  if(!isset($_SESSION['login'])) {
6     if(empty($md5) || (isset($_POST['pass']) && (md5($_POST['pass']) == $md5))) {
7        $_SESSION['login'] = true;
8     }
9     else {
10       die("404 Not Found");
11    }
12 }
```

Listing 5.1: Examplary authentication mechanism.

In order to automatically enumerate activated and deactivated authentication mechanisms, we installed our set of shells in a sandbox and visited their root page. Based upon our manual analysis, the following results were observed for our set of PHP shells:

- 52.0 % provide an authentication mechanism in the code, i. e., 250 out of 481
- 30.8 % of the authentication mechanism can be bypassed, i. e., 77 out of 250
- 28.4 % of the authentication mechanism are activated by default, i. e., 71 out of 250
- 25.4 % of the, by default, activated authentication mechanism can be bypassed, i. e., 18 out of 71

In the following, we present different types of backdoors we encountered in our set of PHP shells that can be used to bypass the authentication mechanism. We believe that these were installed intentionally by the creator or a redistributor of the PHP shell. We also observed that many authentication mechanisms were copied among different shell families and variants, including (perhaps unintentionally) the backdoor code. The detected backdoors can be grouped into three categories: (i) registering global variables, (ii) using unprotected features, and (iii) leaking the authentication credentials. Note that, due to the code complexity and large number of analyzed PHP shells, we do not claim completeness for the detection of all backdoors in our set.

Register Globals

As shown earlier, in Listing 2.7, the dangerous PHP setting `register_globals` can be simulated with a call to a single PHP built-in function. The security implications are subtle and hard to spot for untrained eyes, making them a perfect choice for planting backdoors in PHP shells. In fact, in 70.1 % of the backdoored PHP shells, one of these features was injected between the authentication configuration and the check, allowing to bypass the authentication completely. For example, the authentication mechanism shown in Listing 5.1, when activated and backdoored, could then be bypassed by overwriting the variable `$md5` with an arbitrary password or by setting the `$_SESSION['login']` key directly (`index.php?md5=0&_SESSION[login]=1`). This backdoor is found in every *c99* shell and was copied to a variety of sibling shells which adopted *c99*'s extensive configuration code that includes a subtle call to `extract()`.

Unprotected Features

Other authentication mechanisms could be bypassed during our analysis by abusing unprotected functionality (22.1 %). Sometimes likely by accident, sometimes clearly intentional, certain features are not protected by the authentication mechanism and can be abused by an insider. Listing 5.2 demonstrates three positions within a protected PHP shell where we found backdoor code. For example, we found several code execution vulnerabilities at position one or two, which are accessible before the authentication is performed. This allows an attacker to either upload another shell to the compromised server or to retrieve the shell's source code including its login credentials. Moreover, one shell switched the variable `$s_auth` to true in case the GET parameter *error* was set. Other samples were extended with a new feature at position two or three. Although the corresponding HTML interface was not visible without authentication, the backend of these features could still be accessed and abused by an insider. For example, the feature to download the current directory as a ZIP file allowed us to also retrieve the PHP shell itself, revealing its source code and login credentials.

```
1 // position 1
2 $s_auth = false;
3 if(is_authenticated()) { $s_auth = true; }
4 // position 2
5 if($s_auth) { // protected shell features }
6 // position 3
```

Listing 5.2: Example authentication with backdoor locations.

Information Leakage

Last but not least, some shells allow the circumvention of the authentication mechanism by leaking the authentication credentials. For example, in seven shells, an email is sent to the attacker's mailbox that includes the compromised server's domain, path to the shell, as well as the authentication password. The original code was obfuscated and hidden within the shell's features. This makes it hard to spot this functionality during a code review, specifically when only the authentication mechanism itself is investigated.

```
1 $gh = $_SERVER["HTTP_HOST"];
2 $gh .= $_SERVER["REQUEST_URI"];
3 $mv = "La:$auth_pass, es correcta";
4 mail("report@attack.er", $gh, "$gh $mv");
```

Listing 5.3: Email backdoor leaks the password.

5.3.5 Threats to Validity

There are certain threats to validity of our web shell study. First, although we applied several automated and manual preparation steps to our initial corpus of 1 449 shells and reduced the set to one third, we cannot guarantee the quality of our set. It might still include shells with identical features but different obfuscation techniques or syntax. However, our manual investigation for authentication mechanisms showed that all files in our corpus are at least valid PHP shells.

Next, we have no indication of how prevalent each of these shells are in the wild and how representative our corpus is. While some shells indicated their date of creation in the source code with a time stamp, others do not. Furthermore, we observed several modified versions of shells with an equal time stamp. Thus, we believe that interpretating this meta information would be misleading. Naturally, it is impossible to obtain a complete list of all shells and we performed the largest study on web shells so far.

Furthermore, our feature enumeration is based on code-fingerprints which could, in principle, fail for real-world PHP applications. However, the code of PHP shells is comparably simple (no OOP features, minimum inter-procedural analysis necessary) and we did not encounter false positives during a manual investigation of 50 sample shells. Nonetheless, we found further features that were left out of our evaluation, for example, the ability to crack password hashes, attempts to bypass PHP's `safe_mode`, a HTTP proxy, and *Denial of Service* (DoS) features. We expect these features to be less prominent in PHP shells and hard to fingerprint generically, thus, we decided to not include these into our evaluation.

Last but not least, our manual investigation of authentication mechanisms cannot guarantee completeness. On the one hand, it is possible that we missed ways to bypass the mechanism. On the other hand, it is also plausible that some of the backdoors might have been coding errors. However, we believe that this could only be true for some of the unprotected features we detected. A home phoning of the password or the simulation of the `register_globals` setting, instead, looks clearly suspicious to us.

5.4 Discussion

In this chapter, we used our static analysis prototype in order to study the economy of vulnerabilities in PHP applications from the perspective of developers and attackers. The analyses enabled us to propose answers to our five research questions **Q1**–**Q5**.

First, we empirically analyzed how developers utilize data sanitization and validation mechanisms in practice to prohibit malicious input (**Q1**). We analyzed more than 2.5 million LOC and evaluated the most prevelant security mechanisms for HTML, SQL, and JavaScript markup, mainly type-related (**Q2**). As a result, we learned that specific markup contexts and security mechanisms are more prone to pitfalls than others and require increased attention. For example, we found evidence that single-quoted HTML attributes and JavaScript environments are more likely causing a vulnerability than other markup contexts (**Q3**). Moreover, we observed that although some best-practice guidelines on secure web programming emerged (e. g., recommendations by OWASP [97]), no comprehensive security standards are available for developers. Thus, each developer applies his own favorite security mechanism and programming mistakes occur, specifically in custom regular expressions or string modifications. Our results help us (and other static analysis engineers) to focus on the detection and precise simulation of the most common security mechanisms. We expect that (web) applications implemented in other languages contain similar programming patterns that developers, code auditors, and static analysis engineers need to be aware of. To this end, our results serve as a metric to rank vulnerability reports and to verify the code coverage in our tool.

Furthermore, we presented the first comprehensive analysis of malicious web shells, by compiling a set of 481 PHP shells and using a combination of automated and manual analysis to uncover the shells' visible and invisible features. We showed that modern shells provide a wide range of tools to an attacker, ranging from remote command execution and file browsing, to password brute-forcing and port scanning (**Q4**). In addition, we provide evidence that almost a third of the shells that provide an authentication mechanism contain a hidden backdoor functionality, which attackers can abuse to obtain access to compromised servers (**Q5**). Finally, we argue that a better understanding of malicious web shells will naturally result into designing better protection and detection techniques. Therefore, it is our hope that our work and the corresponding datasets can be used to foster new research in the area of web application malware.

Chapter 6

Conclusion

Nowadays, the requirements of security testing tools for modern PHP applications have changed: diverse language features, applied security mechanisms, and unexplored vulnerability types ousted traditional, easy to spot bugs and lead to new, complex security flaws. Simultaneously, the complexity of modern applications grew to hundreds of thousand lines of code which previous static analysis approaches do not scale to. At the same time, there is a rising demand for the early detection of security vulnerabilities due to increasing attacks [128], costs of data breaches [101], and political obligations [35, 153]. In this thesis, we addressed this problem by exploring the root causes for vulnerabilities in PHP applications, proposing a feasible approach for the automated detection, studying the reliability of available security mechanisms in practice, and revealing post-exploitation steps of attackers.

Summary

In particular, we first surveyed the challenges of analyzing various dynamic languages features, pitfall-prone security mechanisms, as well as complex vulnerability types in PHP applications. Based on the insight that previous work did not, or incompletely, addressed these challenges, we proposed a novel static analysis approach for the automated, efficient, and precise vulnerability detection of 36 vulnerability types. We perform a comprehensive analysis of built-in language features, such as more than a thousand PHP built-in functions with respect to the called arguments. This allows us to accurately analyze the data flow, to detect various sources and sinks, and to analyze sanitization and validation in a more comprehensive way compared to prior work in this area. More specifically, our intra- and inter-procedural analysis bases upon summarized data flows which enable a very efficient vulnerability detection. Our object- and field-sensitive data flow analysis allows us to analyze object-oriented code and to detect exploitable gadget chains for *PHP object injection* vulnerabilities. Moreover, we refine our taint analysis with a context-sensitive string analysis of the current markup context that evaluates the interaction with the prevalent source, sink, security mechanism, encoding, and PHP configuration. Hence, even complex occurrences of vulnerabilities can be detected. A SQL markup parser detects readings and writings to persistent data stores for second-order vulnerability detection.

In a three-staged evaluation, our prototype implementation detected hundreds of previously unknown security vulnerabilities in popular real-world applications with a low false discovery rate and outperformed state-of-the-art tools. Furthermore, our empirical studies revealed pitfall-prone security mechanisms and markup contexts in practice, as well as features and backdoors in popular web shells, and helped to improve our prototype. In summary, we could fulfill our goal to develop a non-annotation based security analysis approach that scales to several hundred of thousands lines of code with a comprehensive coverage of language features and vulnerability types in order to detect complex vulnerabilities in modern PHP applications.

Future Work

The major root cause for false positives in our static analysis approach is a *path-insensitive* taint analysis [14, 33, 36, 159]. While we evaluate path constraints regarding to input validation, we dismiss their interaction and mutually exclusive paths constraints. In the future, our approach could be supplemented with a *string constraint solver* [137, 160]. However, this will pose new challenges regarding the efficient resolvement of constraint values and our tool's performance which have to be explored. Contrarily, we see room for performance improvements. For example, our backwards-directed taint analysis could be guided by a forwards-directed *taint propagation* for path preferences. Besides, we plan to support more popular PHP frameworks. In our current implementation, *template engines* and *query builders* can lead to imprecision and false negatives that we want to eliminate. Thereby, framework-specific configuration settings which can be stored in external resources are a challenging task for static reflection analysis [3, 122, 138]. Also, we aim to add new vulnerability types to our prototype, such as the detection of *privilege escalation vulnerabilities* [90, 119, 126]. For this purpose, our manually detected authentication bypasses in popular web shells can be used as a testing environment. Finally, applying our approach to other programming languages will be an interesting experiment.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Alex Aiken. Moss: A system for detecting software plagiarism, 2005. `https://theory.stanford.edu/~aiken/moss/`.

[2] Jon Allen. Perl Taint Mode. `http://perldoc.perl.org/perlsec.html`, as of January 2016.

[3] Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *IEEE/ACM international Conference on Automated Software Engineering (ASE)*, pages 214–223, 2007.

[4] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering (TSE)*, 36(4), 2010.

[5] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[6] Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna. Multi-Module Vulnerability Analysis of Web-based Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[7] Adam Barth, Joel Weinberger, and Dawn Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security Symposium*, 2009.

[8] Daniel Bates, Adam Barth, and Collin Jackson. Regular Expressions Considered Harmful in Client-side XSS Filters. In *International Conference on the World Wide Web (WWW)*, 2010.

[9] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[10] Jason Bau, Frank Wang, Elie Bursztein, Patrick Mutchler, and John C Mitchell. Vulnerability Factors in New Web Applications: Audit Tools, Developer Selection & Languages. Technical report, Stanford, 2012.

[11] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic web sites. In *International Conference on the World Wide Web (WWW)*, 2002.

[12] Paul Biggar and David Gregg. Static Analysis of Dynamic Scripting Languages. 2009.

[13] Prithvi Bisht and VN Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 23–43. 2008.

[14] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Symposium on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[15] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[16] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *International Conference on Software Engineering (ICSE)*, 2011.

[17] Hristo Bojinov, Elie Bursztein, and Dan Boneh. XCS: Cross Channel Scripting and Its Impact on Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[18] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Symposium on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008.

[19] Nikolay Bryskin. Nikicat's web-malware-collection repository. `https://github.com/nikicat/web-malware-collection/tree/master/Backdoors/PHP`.

[20] Davide Canali and Davide Balzarotti. Behind the Scenes of Online Attacks: an Analysis of Exploitation Behaviors on the Web. In *Symposium on Network and Distributed System Security (NDSS)*, 2013.

[21] Davide Canali, Davide Balzarotti, and Aurélien Francillon. The role of web hosting providers in detecting compromised websites. In *International Conference on the World Wide Web (WWW)*, pages 177–188, 2013.

[22] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *International Static Analysis Symposium (SAS)*, 2003.

[23] Steve Christey and Robert A. Martin. Vulnerability Type Distributions in CVE, May 2007.

[24] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[25] Juan José Conti and Alejandro Russo. A taint mode for python via a library. In *Nordic Conference on Secure IT Systems (NORDSEC)*. 2010.

[26] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[27] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[28] Adrian Crenshaw. Web Shells and RFIs Collection. `http://www.irongeek.com/i.php?page=webshells-and-rfis`.

[29] Johannes Dahse. Joomla! 3.0.2 POI (CVE-2013-1453) – Gadget Chains. `https://websec.wordpress.com/2014/10/03/joomla-3-0-2-poi-cve-2013-1453-gadget-chains/`, as of January 2016.

[30] Johannes Dahse and Thorsten Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[31] Johannes Dahse and Thorsten Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *USENIX Security Symposium*, 2014.

[32] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[33] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

[34] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-oriented Programs using Static Class Hierarchy Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–101, 1995.

[35] Deutscher Bundestag. Bundestag beschließt das IT-Sicherheitsgesetz. `https://www.bundestag.de/dokumente/textarchiv/2015/kw24_de_it_sicherheit/377026`.

[36] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, 2008.

[37] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[38] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2010.

[39] Maureen Doyle and James Walden. An Empirical Study of the Evolution of PHP Web Application Security. In *Security Measurements and Metrics (Metrisec)*, 2011.

[40] Manuel Egele, Martin Szydlowski, Engin Kirda, and Christopher Kruegel. Using Static Program Analysis to Aid Intrusion Detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2006.

[41] Stefan Esser. Shocking News in PHP Exploitation. In *Power of Community (POC)*, 2009.

[42] Stefan Esser. Utilizing Code Reuse Or Return Oriented Programming in PHP Applications. In *BlackHat USA*, 2010.

[43] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[44] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive Type Qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[45] The PHP Group. PHP: Extension Membership. `http://php.net/manual/en/extensions.membership.php`, as of January 2016.

[46] The PHP Group. PHP: Manual Quick Reference. `http://php.net/quickref.php`, as of January 2016.

[47] The PHP Group. PHP: Using Register Globals. `http://php.net/manual/en/security.globals.php`, as of January 2016.

[48] Salvatore Guarnieri and V Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, pages 151–168, 2009.

[49] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from Vulnerable JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[50] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[51] William GJ Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *IEEE/ACM international Conference on Automated Software Engineering (ASE)*, pages 174–183, 2005.

[52] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL Injection Attacks and Countermeasures. In *IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[53] David Hauzar. Towards Static Analysis of Languages with Dynamic Features. *Doctoral Thesis*, pages 99–100, 2014.

[54] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *ACM Symposium On Applied Computing (SAC)*, pages 1663–1671, 2014.

[55] Mark Hills, Paul Klint, and Jurgen Vinju. An Empirical Study of PHP Feature Usage. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2013.

[56] Chad Holmes. Malware Lateral Movement: A Primer. `https://www.fireeye.com/blog/executive-perspective/2015/08/malware_lateral_move.html`, as of January 2016.

[57] Peter Hooimeijer. Decision Procedures for String Constraints. *Ph.D. Dissertation, University of Virginia*, 2010.

[58] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*, 2011.

[59] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *International Conference on the World Wide Web (WWW)*, pages 148–159, 2003.

[60] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Verifying Web Applications Using Bounded Model Checking. In *Conference on Dependable Systems and Networks (DSN)*, 2004.

[61] 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 *International Conference on the World Wide Web (WWW)*, 2004.

[62] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for JavaScript. In *ACM Symposium On Applied Computing (SAC)*, pages 1930–1937. ACM, 2009.

[63] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *International Conference on the World Wide Web (WWW)*, 2007.

[64] Bryant Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[65] Nenad Jovanovic. TUVSA-0603-002 - MyBloggie: Multiple XSS Vulnerabilities. `http://www.iseclab.org/advisories/TUVSA-0603-002.txt`, as of January 2016.

[66] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy (S&P)*, 2006.

[67] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[68] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications. *Journal of Computer Security (JCS)*, 18(5), 08 2010.

[69] Brian W. Kernighan and Rob Pike. The Practice of Programming. In *Addison-Wesley, Inc*, 1999.

[70] Nidal Khoury, Pavol Zavarsky, Dale Lindskog, and Ron Ruhl. Testing and Assessing Web Vulnerability Scanners for Persistent SQL Injection Attacks. In *Proceedings of the First International Workshop on Security and Privacy Preserving in e-Societies*, SeceS '11, pages 12–18, 2011.

[71] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *International Conference on Software Engineering (ICSE)*, 2009.

[72] Jinsuk Kim, Dong-Hoon Yoo, Heejin Jang, and Kimoon Jeong. WebSHArk 1.0: A Benchmark Collection for Malicious Web Shell Detection. In *Journal of Information Processing Systems (JIPS)*, 2015.

[73] Amit Klein. Cross-Site Scripting Explained. *Sanctum White Paper*, 2002.

[74] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Phantm: PHP Analyzer for Type Mismatch. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2010.

[75] Eddie Kohler. HotCRP Conference Management Software. `http://www.read.seas.harvard.edu/~kohler/hotcrp/`, as of January 2016.

[76] Jesse Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digital investigation*, 3:91–97, 2006.

[77] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1193–1204, 2013.

[78] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[79] Benjamin Livshits and Stephen Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013.

[80] Benjamin Livshits and Weidong Cui. Spectator: Detection and Containment of JavaScript Worms. In *USENIX Annual Technical Conference*, 2008.

[81] Benjamin Livshits and Monica S Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[82] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems (APLAS)*, pages 139–160, 2005.

[83] Mango. phpMyAdmin 3.x Multiple Remote Code Executions. `http://ha.xxor.se/2011/07/phpmyadmin-3x-multiple-remote-code.html`, as of January 2016.

[84] Sean McAllister, Engin Kirda, and Christopher Kruegel. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[85] Microsoft Developer Network Library. Naming Files, Paths, and Namespaces. `http://msdn.microsoft.com/en-us/library/aa365247(VS.85)`, as of January 2016.

[86] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. *ACM SIGSOFT Software Engineering Notes*, 27(4):1–11, 2002.

[87] Yasuhiko Minamide. Static Approximation of Dynamically Generated Web Pages. In *International Conference on the World Wide Web (WWW)*, 2005.

[88] MITRE. Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org/`, as of January 2016.

[89] Mattia Monga, Roberto Paleari, and Emanuele Passerini. A hybrid analysis framework for detecting web application vulnerabilities. In *ICSE Workshop on Software Engineering for Secure Systems (SESS)*, pages 25–32, 2009.

[90] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. Mace: Detecting privilege escalation vulnerabilities in web applications. In *ACM Conference on Computer and Communications Security (CCS)*, pages 690–701, 2014.

[91] MyBB. Open Source Discussion Board. http://www.mybb.com/, as of January 2016.

[92] myWebland Group. myBloggie Weblog System. http://mybloggie.mywebland.com/, as of January 2016.

[93] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *International Conference on Software Engineering (ICSE)*, pages 580–586, 2005.

[94] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[95] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference (SEC)*, 2005.

[96] osCommerce. Creating Online Stores Worldwide. http://www.oscommerce.com/, as of January 2016.

[97] OWASP. OWASP Secure Coding Practices. https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide, as of January 2016.

[98] The Open Web Application Security Project (OWASP). List of Vulnerabilities. https://www.owasp.org/index.php/Category:Vulnerability, as of January 2016.

[99] PHP-Nuke. CMS Portal Solution. http://www.phpnuke.org/, as of January 2016.

[100] phpBB. Free and Open Source Forum Software. http://www.phpbb.com/, as of January 2016.

[101] Ponemon Institute. 2015 Cost of Data Breach Study: Global Analysis. 2015.

[102] Ben Potter and Gary McGraw. Software security testing. *IEEE Symposium on Security and Privacy (S&P)*, 2(5):81–85, 2004.

[103] Donald Ray and Jay Ligatti. Defining Code-Injection Attacks. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2012.

[104] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), March 2012.

[105] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for Javascript. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[106] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[107] Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[108] Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *ACM Symposium On Applied Computing (SAC)*, 2012.

[109] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[110] R Sekar. An Efficient Black-Box Technique for Defeating Web Application Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[111] SektionEins GmbH. Piwik Cookie Unserialize() Vulnerability. `https://www.sektioneins.de/en/advisories/advisory-032009-piwik-cookie-unserialize-vulnerability.html`, as of January 2016.

[112] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. *USENIX Security Symposium*, pages 201–220, 2002.

[113] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting Common Web Application Vulnerabilities from Input Validation and Sanitization Code Patterns. In *IEEE/ACM international Conference on Automated Software Engineering (ASE)*, 2012.

[114] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C Briand. Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis. In *International Conference on Software Engineering (ICSE)*, 2013.

[115] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[116] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick Your Contexts Well: Understanding Object-sensitivity. *ACM Symposium on Principles of Programming Languages (POPL)*, 2011.

[117] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 248–259, 2015.

[118] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`, as of January 2016.

[119] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. RoleCast: Finding Missing Security Checks when You Do Not Know What Checks Are. In *ACM SIGPLAN*

Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2011.

[120] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1181–1192, 2013.

[121] Sooel Son and Vitaly Shmatikov. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2011.

[122] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: Taint Analysis of Framework-based Web Applications. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

[123] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation Tracking for Points-to Analysis of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458. 2012.

[124] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

[125] Fangqi Sun, Liang Xu, and Zhendong Su. Client-side Detection of XSS Worms by Monitoring Payload Propagation. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[126] Fangqi Sun, Liang Xu, and Zhendong Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *USENIX Security Symposium*, 2011.

[127] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical Virtual Method Call Resolution for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

[128] Symantec. Internet Security Threat Report, Volume 20. 2015.

[129] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[130] Tennc. Webshell repository. `https://github.com/tennc/webshell/`.

[131] The PHP Group. History of PHP. `http://php.net/manual/en/history.php.php`, as of January 2016.

[132] The PHP Group. PHP: Autoloading Classes. `http://www.php.net/manual/language.oop5.autoload.php`, as of January 2016.

[133] The PHP Group. PHP: Magic Methods. `http://www.php.net/manual/language.oop5.magic.php`, as of January 2016.

[134] Dave Thomas and Andy Hunt. Programming Ruby: A Pragmatic Programmer's Guide. *Addison-Wesley*, 2000.

[135] Stephen Thomas, Laurie Williams, and Tao Xie. On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities. *Information and Software Technology*, 51(3):589–598, 2009.

[136] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[137] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1232–1243, 2014.

[138] Omer Tripp, Marco Pistoia, Stephen Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2009.

[139] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[140] John Troony. php-webshells repository. `https://github.com/JohnTroony/php-webshells`.

[141] Truong Dinh Tu, Cheng Guang, Guo Xiaojun, and Pan Wubin. Webshell detection techniques in web applications. In *Computing, Communication and Networking Technologies (ICCCNT), 2014 International Conference on*, pages 1–7. IEEE, 2014.

[142] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[143] UnPHP. The Online PHP Decoder. `http://www.unphp.net/`.

[144] UtopiaSoft. Utopia News Pro. `http://www.utopiasoftware.net/newspro/`, as of January 2016.

[145] Victor van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2012.

[146] W3Techs. Usage of Content Management Systems for Websites. `http://w3techs.com/technologies/overview/content_management/all`, as of January 2016.

[147] W3Techs. Usage of Server-side Programming Languages for Websites. `http://w3techs.com/technologies/overview/programming_language/all`, as of January 2016.

[148] James Walden, Adam Messer, and Alex Kuhl. Measuring the Effect of Code Complexity on Static Analysis Results. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2009.

[149] Gary Wasserman and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[150] Gary Wasserman and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering (ICSE)*, 2008.

[151] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 249–260, 2008.

[152] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[153] White House. Executive Order – Improving Critical Infrastructure Cybersecurity. `https://www.whitehouse.gov/the-press-office/2013/02/12/executive-order-improving-critical-infrastructure-cybersecurity`.

[154] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, 2006.

[155] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

[156] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. STRANGER: An Automata-based String Analysis Tool for PHP. In *Symposium on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[157] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching Vulnerabilities with Sanitization Synthesis. In *International Conference on Software Engineering (ICSE)*, 2011.

[158] Yunhui Zheng and Xiangyu Zhang. Static Detection of Resource Contention Problems in Server-side Scripts. In *International Conference on Software Engineering (ICSE)*, pages 584–594, 2012.

[159] Yunhui Zheng and Xiangyu Zhang. Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection. In *International Conference on Software Engineering (ICSE)*, 2013.

[160] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pages 114–124, 2013.

Johannes Dahse

Personal Data

Date of Birth:	27. 04. 1987
Place of Birth:	Jena, Germany
Email:	johannes.dahse@rub.de

Education

1997 - 2006	Angergymnasium Jena
2006 - 2012	Study of *IT Security* (Diplom), **Ruhr-University Bochum**
2013 - 2016	PhD candidate, Chair for Systems Security, **Ruhr-University Bochum**

Work experience

2011	Security Consultant for **Sektion Eins GmbH**, Cologne, Germany
2011 - 2012	Security Consultant for **SEC Consult**, Vienna, Austria
2012	Security Researcher for **Qualys, Inc.**
since 2012	Security Consultant for **Cure53 GmbH**, Berlin, Germany

Awards

Aug 2014	**Internet Defense Prize** from Facebook at USENIX Security *Static Detection of Second-Order Vulnerabilities in Web Applications*
Nov 2014	**Best Student Paper Award** at the ACM CCS conference *Code Reuse Attacks in PHP: Automated POP Chain Generation*