UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Monitoring Web Applications for Vulnerability Discovery and Removal Under Attack

Paulo David Ferreira Antunes

Mestrado em Engenharia Informática
Especialização em Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves
Prof. Doutora Ibéria Vitória de Sousa Medeiros

2018

Acknowledgments

First of all, I would like to express my gratitude towards my advisors, Prof. Nuno Neves and Prof. Ibéria Medeiros, their guidance and feedback were essential for me to keep focused on the key aspects of the dissertation.

I would also like to thank Inês Gouveia, Ricardo Costa, Henrique Mendes, Joaquim Afonso, Tomás Peixinho and Miguel Falé. I had the pleasure of sharing my academical course with all of you and I could not have asked for better company and work groups during difficult situations and stressful times. To my childhood friends, João Figueira and Gonçalo Catarino, thank you for all your support over the years and for providing me with moments of respite when I needed them the most.

Finally, I would like to thank my family, for always encouraging me to give my best in my studies and motivating me to tackle on new challenges.

This work was partially supported by the EC through project FP7 SEGRID (607109), by the FCT through the project SEAL (PTDC/CCI-INF/29058/2017), and LASIGE Research Unit (UID/CEC/00408/2013).

Resumo

A rápida evolução das tecnologias web juntamente com o acesso fácil a poder computacional levou a que vários serviços passassem a dispor de uma vertente online. Por exemplo, através de serviços web pode-se realizar comodamente compras online, contactar amigos ou aceder a contas bancárias. Todas estas funcionalidades são geralmente concretizadas através de aplicações web e, embora sejam utilizadas por milhares de milhões de utilizadores diariamente, muitas delas incorporam vulnerabilidades latentes. Se exploradas, as falhas advindas delas poderão levar a diversas consequências que incluem o roubo de dados privados, a adulteração de informação, e a inoperabilidade dos serviços, resultando em grandes prejuízos para utilizadores e para empresas. A existência de vulnerabilidades prende-se com práticas de programação insegura aliadas à utilização de linguagens e de sistemas gestores de conteúdos (ex., WordPress, Drupal) com poucas validações. Por exemplo, o PHP tem limitações na validação de tipos, mas acaba por ser a linguagem mais utilizada no desenvolvimento de aplicações web.

De entre as classes de vulnerabilidade presentes em aplicações web, as de injeção de código continuam a ser as mais prevalecentes, sendo colocadas pela OWASP no topo da lista de 2017. As vulnerabilidades de injeção de SQL (SQLI) pertencem a esta categoria, e se exploradas levam a consequências nefastas quer aos serviços como às bases de dados. Por exemplo, recentemente, um ataque de SQLI possibilitou o acesso criminoso a dados de cartões de crédito, em que os seus danos já atingiram os 300 milhões de dólares.

Apesar da classe de vulnerabilidade Cross-Site Scripting (XSS) aparecer em sétimo lugar na lista da OWASP, estima-se que estas faltas estejam presentes em cerca de dois-terços das aplicações web. Tanto que 2017 foi considerado como sendo o ano da ”onda XSS”, tendo havido a expetativa de um crescimento de 166% face ao ano de 2016. Este crescimento acabou por se efetivar de acordo com a Imperva.

As vulnerabilidades podem ser encontradas no código das aplicações recorrendo-se a ferramentas de análise estática de código. A técnica que tem sido mais utilizada é a análise de comprometimento, que rastreia os inputs (maliciosos) ao longo do código da aplicação e verifica se estes atingem alguma função da linguagem que pode ser explorada (ex., a função *echo* em PHP). A precisão destas ferramentas depende do conhecimento que elas possuem sobre as classes de vulnerabilidades a procurar, bem como na implementação dos mecanismos que elas empregam na análise do código, que se incompleta ou incorreta

pode originar falsos positivos (indicação de vulnerabilidade que não existem) ou falsos negativos (a não deteção de vulnerabilidades). Por outro lado, a deteção por si só é insuficiente, uma vez que o objetivo final é a remoção das vulnerabilidade, algo que poderá ser complicado de realizar por programadores pouco familiarizados com aspetos de segurança.

Esta dissertação apresenta uma solução para a deteção e remoção automática de vulnerabilidades em aplicações web, baseada numa *análise estática de código direcionada*. Através da monitorização da aplicação em execução e das interações com ela, são identificados os inputs potencialmente maliciosos. Estes inputs são explorados para direcionar a análise estática, permitindo comprovar eventuais vulnerabilidades e removê-las automaticamente através da inserção de *fragmentos de código* na aplicação, gerando uma versão nova e mais segura. De forma a concretizar este objetivo pressupõe-se a execução das seguintes tarefas.

Primeiramente, é feito um levantamento e estudo de algumas vulnerabilidades que sendo exploradas poderão levar a consequências severas, que são facilmente exploráveis ou que estão presentes num grande volume das aplicações web modernas. Esta análise visa clarificar e exemplificar as vulnerabilidades elucidando a forma como estas surgem, como podem ser exploradas de forma maliciosa por atacantes e como podem ser corrigidas ou evitadas.

Em segundo lugar, são analisadas várias técnicas e as ferramentas que existem atualmente para a deteção de vulnerabilidades. Este estudo foca-se primariamente em ferramentas de análise estática, fuzzing, execução simbólica e oráculos com o objetivo de compreender as suas várias formas de implementação bem como os seus pontos fortes e fraquezas de forma a compreender o que pode ser melhorado ou acrescentado.

Em terceiro lugar, é feita uma análise de indicatores de ataque e de comprometimento. Estes indicadores são relevantes no sentido em que permitem que o sistema esteja ciente de uma tentativa de ataque, permitindo-lhe tomar medidas proativas ou que seja confirmado que de facto houve um ataque ao sistema.

Em quarto lugar, conjugando todos os estudos anteriores é desenvolvida uma arquitetura que é capaz de detetar e corrigir vulnerabilidades durante a execução da aplicação web recorrendo a *análise estática de código direcionada* e a *fragmentos de código*, respetivamente.

Finalmente, é implementado um protótipo da arquitetura proposta e uma avaliação do mesmo com 5 aplicações web em execução, o qual se mostrou eficaz na deteção e correção de vulnerabilidades de XSS e SQLI. A ferramenta detetou 174 vulnerabilidades, onde 2 delas ainda não tinham sido descobertas (i.e., ”zero-days”), e não registou falsos positivos.

Palavras-chave: vulnerabilidades, aplicações web, segurança de software, análise estática de código direcionada, correção de código

Abstract

Web applications are ubiquitous in our everyday lives, as they are deployed in the most diverse contexts and support a variety of services. The correctness of these applications, however, can be compromised by vulnerabilities left in their source code, often incurring in nefarious consequences, such as the theft of private data and the adulteration of information.

This dissertation proposes a solution for the automatic detection and removal of vulnerabilities in web applications programmed in the PHP language. By monitoring the user interactions with the web applications with traditional attack discovery tools, it is possible to identify malicious inputs that are eventually provided by attackers. These inputs are then explored by a *directed static analysis* approach, allowing for the discovery of potential security issues and the correction of bugs in the program.

The solution was implemented and validated with a set of vulnerable web applications. The experimental results demonstrate that the tool is capable of detecting and correcting SQL Injection and XSS vulnerabilities. In total 174 vulnerabilities were found in 5 web applications, where 2 of these were previously unknown by the research community (i.e., they were ”zero-day” vulnerabilities).

Keywords: vulnerabilities, web applications, directed static analysis, code correction, software security

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface. 20

AST Abstract Syntax Tree. 12, 20, 46, 55, 56

BP Blog Preprocessor. 24

BS Blog Scraper. 24

CPU Central Processing Unit. 22

CRS Core Rule Set. 39, 41, 48

CSP Content Security Policy. 25

DBMS Database Management System. 23, 47

DFA Deterministic Finite-state Automatons. 18

DOM Document Object Model. 10, 13, 26

DTG Dynamic Test Generation. 18

EGT Execution-Generated Testing. 19

GSE Generalised Symbolic Execution. 18

GUI Graphical User Interface. 16

GWFG General Workflow Graph. 21

HTML Hypertext Markup Language. 31

HTTP Hypertext Transfer Protocol. 21, 22, 42, 44, 48

IoA Indicator of Attack. 23, 25, 32, 41

IoC Indicator of Compromise. 23–25

IP Internet Protocol. 11, 33, 42

MD5 Message Digest 5. 57

MLFA Multi-Level Finite Automat. 18

OS Operating System. 8

PHP PHP: Hypertext Preprocessor. 1, 12–14, 20, 35, 44, 46, 47, 55, 56, 58

RC Relation Checker. 24

RCP Relevant Content Picker. 24

SMTP Simple Mail Transfer Protocol. 8

SOAP Simple Object Access Protocol. 8

SQL Structured Query Language. 8, 20–23, 31, 37, 39, 43, 47, 48, 51, 54–57

SQLI SQL Injection. 2, 15, 23, 41, 50, 51, 53–55, 57

SSA Single Static Assignment. 13

TEPT Tainted Execution Path Tree. 13

TST Tainted Symbol Table. 13

UDA User Data Access. 22

UML Unified Modelling Language. 41, 43

URI Uniform Resource Identifier. 9, 33, 43, 50

URL Uniform Resource Locator. 22, 24–26, 33

VM Virtual Machine. 24, 53, 54

WAF Web Application Firewall. 39

XML eXtensible Markup Language. 8, 9, 24

XSS Cross-Site Scripting. 2, 3, 9–11, 13, 16, 17, 20, 25, 31, 39, 41, 49–51, 53, 55, 56

XXE XML External Entity. 8, 9

Chapter 1

Introduction

The quick evolution of web technologies allied with the easy access to computational power has lead many services to have an online component. Through web services it is possible to easily purchase various products, contact friends and family or even access bank accounts. All these functionalities are generally provided through web applications and, although they are used by billions of users on a daily basis, many of them incorporate latent vulnerabilities. If exploited, these mistakes can lead to various consequences that include theft of private data, adulteration of information and unavailability of services. In some cases the vulnerability might not only jeopardize just a single user but the entire service, to the point that it might be completely taken over by an adversary, resulting in huge financial losses for both users and companies. The existence of these vulnerabilities steams from insecure programming practices paired with the utilization of languages and content management systems (e.g., WordPress, Drupal) with few validation procedures [29] [47]. For example, although PHP has limitations in data type validation, it is still the most used language for the development of web applications [46].

One of the commonly used methods to detect vulnerabilities is resorting to static analysis tools, which can be run at any stage of the life cycle of an application [31] [40] [37]. There are several techniques that can be used to inspect the code statically, however the one that is the most usual is taint analysis, which tracks malicious inputs provided to the application interface (*entry points*) through the code and verifies if they reach any function of the language that can be explored, often designated as *sensitive sinks* (e.g., *echo* function in PHP). The precision of these tools depends on the knowledge they possess about the vulnerability classes, as well as the implementation of mechanisms that they employ to analyse the code, which if incomplete or incorrect may originate false positives (indication of an alarm for non-existent vulnerabilities) or false negatives (not detecting a vulnerability). On the other hand, the detection itself is insufficient, since the final objective is the removal of these vulnerabilities, something that is hard to accomplish by programmers that are not familiarized with security issues.

1.1 Motivation

There are several threats towards web applications that are actively being exploited at this moment. The approaches that exist nowadays to address this problem employ techniques such as *fuzzing* that provides malicious input to an application in order to discover vulnerabilities by trying to force erroneous behaviour. In alternative, *static analysis* examines the code of the target application to find attacker controlled data that reaches sensitive sinks. *Symbolic execution* maps the program state symbolically in order to find bugs, and the usage of *oracles* allows the checking of requests of the web application to a data storage to determine if an attack is being attempted or not.

Considering the existing solutions, we believe that they have limitations: *fuzzing* only sends malicious and/or erroneous input to the application to force a vulnerability and does not provide an accurate description of how it occurs; *static analysis* tools have their precision dependent on the vulnerability classes knowledge base as well as on the mechanisms they utilize to check the code, which, if done incorrectly, can lead to false positives and negatives; *symbolic execution* is usually time consuming and several tools have a significant amount of false positives; *oracles* only indicate that there is a malicious request flowing from the web application to the data storage and do not provide an in-depth description of how the vulnerability was exploited.

Besides these limitations, all of them lack the ability to correct the vulnerabilities automatically. This requires a programmer to dedicate time and effort to remove the flaws within the code. However, this task can be arduous, especially if the programmer is not familiarized with security aspects, which can lead to ineffective measures or even the introduction of other problems within the code.

The goal of this thesis is to detect vulnerabilities by monitoring the interactions with the web application at runtime, in order to extract both input and output information that can be used to effectively scan the code of the web application via a *directed static analysis*. It also removes the vulnerabilities found by inserting *code fragments* in the program according to the detected problem.

The thesis focuses on two classes of vulnerabilities, SQL Injection (SQLI) and Cross-Site Scripting (XSS). Of all vulnerability classes that may appear in web applications, the code injection class continues to be the most prevalent. In fact, it is placed by OWASP at the top of the 2017 list of most relevant vulnerabilities [49]. SQLI belongs to this category, and if exploited it can lead to nefarious consequences both to the services and the databases. An SQLI vulnerability can potentially provide to an attacker complete control over a backend database, such as full access to user credentials, deleting entire databases, creating false entries or altering critical data. For example, a recent SQLI attack allowed criminal access to credit card information, in which the estimated loss has already reached 300 million dollars [39].

Even though the XSS vulnerability appears in the 7th place of the list by OWASP, it

is estimated that these flaws are present in about two-thirds of all web applications. 2017 was considered the year of the ”XSS wave”, with a significant growth of the reported problems due to this type of bug [29]. The impact of XSS is typically smaller than SQLI, but since it is widespread it affects multiple users. There is also an extra problem, a XSS vulnerability might allow an attacker to obtain client credentials for a specific site, which the user shares with other websites. As a consequence, those accounts may also be compromised.

1.2 Objectives

This document studies relevant vulnerabilities in web applications along with the current forms of detection. It also presents a new tool that is capable of discovering and correcting vulnerabilities at runtime. As such, this dissertation has the following goals:

- Study some of the most important and widespread web application vulnerabilities, understanding how they appear in the programs, how they can be fixed and what are the consequences that they can lead to if exploited.

- Investigate current solutions to detect vulnerabilities, focusing on the strengths and weaknesses of each type of approach while generating an understanding of how they can be improved. Also, study various ways of how an attack can be represented, what indicators can prove that a system has been compromised and what are the best forms of correcting the application code depending on the type of vulnerability.

- Design an architecture that is capable of detecting vulnerabilities in web applications and correcting them automatically at runtime, without requiring human intervention over the code. Implement a prototype of the solution and experimentally evaluate the tool with open-source web applications.

1.3 Contributions

The main contributions of the thesis are:

- An architecture that can be used to detect and patch vulnerabilities in web applications at runtime. The proposed architecture takes advantage of rule matching (i.e., via sniffers or firewalls) to identify potentially malicious user inputs provided to the web application. It configures a static analyser to only taint the fields included in the request sent by the client, using it to only follow the code paths that would be compromised by the user inputs. Moreover, the proposed architecture has the

capability to patch the vulnerabilities that are found during static analysis by inserting code fragments in the application program, which employ sanitization and validation procedures.

- Build a proof of concept through the implementation of a working prototype that displays: (1) the capability to detect possible malicious inputs that are being provided to the application by using rule matching to detect anomalous activity; (2) use these malicious inputs to *direct the static analysis* to explore the paths of the code that these inputs would reach; (3) utilize the results of the static analysis to confirm or deny the existence of vulnerabilities related to these inputs; (4) automatically correct the vulnerabilities found during static analysis; (5) maintain a log of the identified vulnerabilities and an indication if the rule matching generated a false positive.

- Carry out an experimental evaluation of the currently implemented prototype with five web applications, which reveals the viability and effectiveness of the developed tool. Overall, a large number of vulnerabilities were discovered, where two of them were zero-day vulnerabilities.

The developed work supported the publication of a full paper (12 pages) at the INForum 2018, in the track of Security of Computational Systems:

P. Antunes, I. Medeiros and N. Neves, Remoção Automática de Vulnerabilidades usando Análise Estática de Código Direcionada, at the Simpósio Nacional de Informática (INForum), September 2018

1.4 Thesis Structure

This thesis is structured as follows:

Chapter 2 briefly explains some relevant concepts and provides fundamental context for the work. Classes of vulnerabilities are described in more detail, along with many of the techniques used for vulnerability detection, such as: static analysis, fuzzing, symbolic execution and oracles. Representations of indicators of attacks and compromise are also studied as they allow for awareness of a possible threat and confirmation of a successful attack respectively.

Chapter 3 presents the proposed architecture, describing the main components as well as how they interact with each other. Furthermore, alternative specialized architectures are also discussed and explained.

Chapter 4 describes the current implementation and proof of concept of the proposed architecture, diving into more details of each module.

Chapter 5 evaluates and validates the implemented tool with five web applications.

Finally, Chapter 6 provides a conclusion of the developed research and discusses future work that can be built upon the base architecture.

Chapter 2

Context and Related Work

This chapter provides some important context that serves as a basis for the research and it also describes some related work. Section 2.1 explains several vulnerability classes that are referenced by OWASP as being among the most critical [49]. Section 2.2 explores some techniques used in static analysis. Section 2.3 looks into fuzzing techniques and various fuzzer tools. Section 2.4 describes how symbolic execution can be utilized to find vulnerabilities. Section 2.5 presents related work about oracles. Lastly, Section 2.6 gives a brief look into indicators of compromise and indicators of attack, alongside tools and frameworks developed for their support.

2.1 Vulnerabilities

Vulnerabilities are bugs that are accidentally introduced during application development. An important note is that not all bugs are vulnerabilities, as only those that might be exploited or used in a way that jeopardizes the security of a system can be considered a vulnerability. For example, a programming error that causes a sentence to lack one space is different from a coding flaw that causes the system crash. In this case, the former is merely a bug and the latter is a vulnerability if the crash can be forced to occur by an adversary. There are many kinds of vulnerabilities, and therefore organizations have created rankings that order classes of vulnerability by relevance, such as the OWASP Top 10 Most Critical Web Application Security Risks [49]. To calculate these risks, various factors are considered, namely exploitability, weakness prevalence, weakness detectability and technical impacts. We will take a look at these critical security risks and analyse their ranking along with the reasons that cause them to be so significant. From this Top 10, the following are issues that could be addressed with the monitoring tool to be developed in this work.

2.1.1 Injection

Injection vulnerabilities allow adversaries to provide malicious code or other inputs to a web application that then gives it to another system. These attacks can exploit for example, system calls, program execution and calls to databases (i.e., a SQL Injection). This type of flaws can range from being easy to discover and exploit to being hidden deep within the system.

In more detail, an injection flaw occurs when an attacker is able to send hostile data to an interpreter employed by the web application. In this case, any source of data can be used as an injection vector, like usernames, parameters and internal or external web services. Injection flaws are quite common, especially in legacy code. They are often found in SQL and NoSQL queries, but can also be occur in OS commands, XML parsers, SMTP Headers, expression languages and others.

This type of vulnerability is typically easy to discover, particularly when there is access to the code. However, even without access to the code, adversaries can try blind attacks manually or use fuzzers/active scanners to find the vulnerability. Injections can have devastating consequences such as data loss, corruption, denial of access or even the release of private information from users. In a worst case scenario an injection can lead to a complete host takeover.

Figure 2.1 illustrates a basic SQL Injection in which the attacker sends a tautology (a condition that is persistently true) to the web application, which allows him to retrieve information from every single user existent in a database, managed by a database management system (DBMS). First the attacker sends a request which is forged to exploit the lack of character escaping, this means that any additional condition can be given after the `"` sign. The adversary utilizes this opportunity to provide the tautology (`or 1=1` is systematically true) after the first condition (`user="user1"`) is provided. Once the DBMS processes this query, when selecting the users from the database it will select them based on whether they fulfil either the first condition or the tautology (due to the usage of the `OR` operator). Since the tautology is in effect, every registered user in the database will match the condition and have their data sent to the attacker. This type of vulnerability can be treated by sanitizing input properly so that injections cannot be executed within the DBMS.

2.1.2 XML External Entity (XXE) Processing

An XXE processing is a type of vulnerability that appears in applications that parse XML inputs. More concretely it happens when an XML input that contains a reference to an external entity is processed by a usually poorly configured XML parser. Adversaries can exploit this vulnerability by accessing web pages or web services, particularly SOAP services that process XML. Most legacy XML processors are susceptible, as they allow for

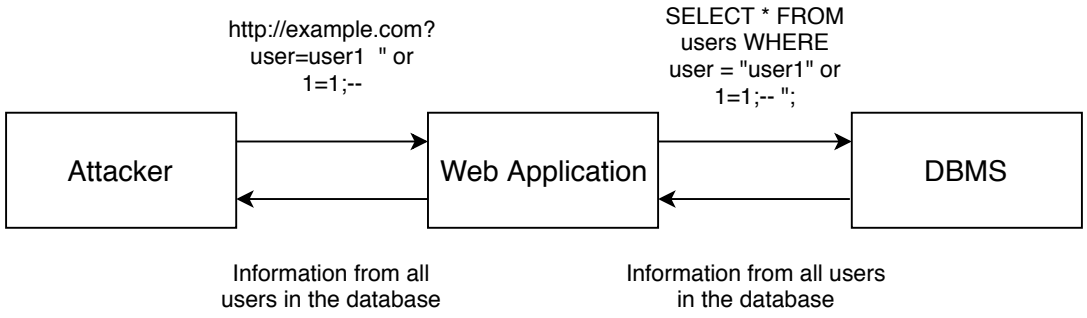

Figure 2.1: Example of a SQL injection attack.

an external entity to be specified, such as a URI that is dereferenced and evaluated during XML processing. XXE processing can lead to data extraction, remote request execution, system scanning, denial of service, among others. Static analysis can discover this issue by looking into dependencies and configurations. A possible manner of dealing with this situation would be to keep parsers well configured and have them check whenever untrusted external entities are invoked.

2.1.3 Cross-Site Scripting (XSS)

Cross-site scripting is a type of vulnerability that allows the attacker to execute a script in the browser of the victim. There are three different types of XSS attacks:

1. Reflected XSS (non-persistent): A reflected XSS occurs when an attacker is able to trick the victim to send a script to a vulnerable web application which then returns it back to the victim's browser. For this to happen the user usually needs to interact with a link or some other attacker controlled page.

Figure 2.2 presents a typical XSS scenario, an attacker forges a URL that will include the malicious script, he then tricks the victim to use this URL (e.g., via *spear phishing*). The erroneous URL provided by the attacker with include the script in the vulnerable application. Finally, the browser of the victim accepts the page provided by the web application and executes the script which will exploit the vulnerability and send information about the user to the attacker (e.g., user credentials).

2. Stored XSS (persistent): A stored XSS happens when a web application stores unsanitized input, which later that can be accessed by another user via a browser. The browser will receive and then execute the stored malicious script.

Figure 2.3 portrays a typical stored XSS case. Initially, an attacker exploits the lack of sanitization of stored input and places some script within the application (e.g., a data file of a forum application). When a victim accesses the compromised data through a browser it will include the script that will be executed and provide the attacker with user credentials or the chance to hijack the current session.

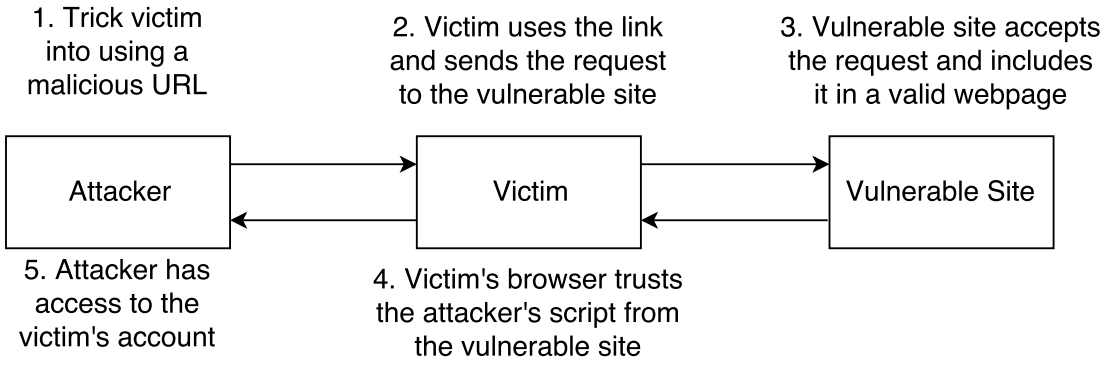

Figure 2.2: Example of a reflected XSS attack.

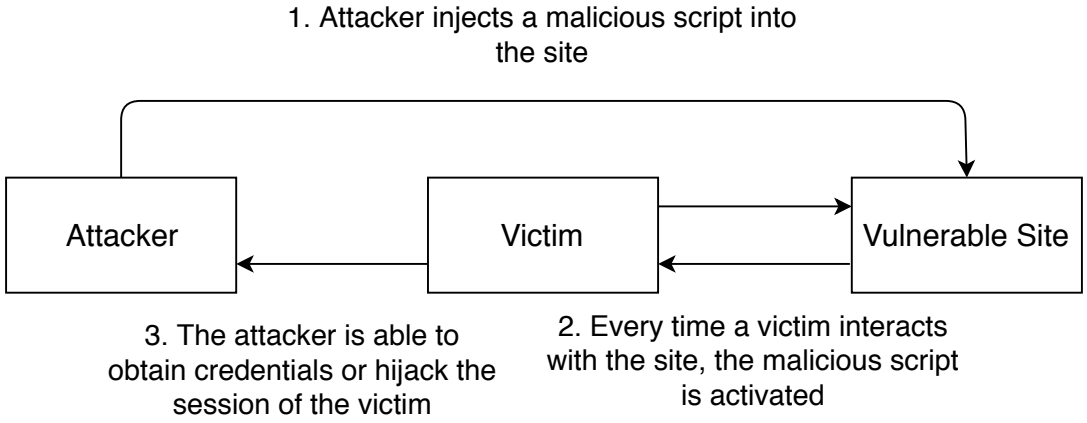

Figure 2.3: Example of a stored XSS attack.

3. DOM (Document Object Model) XSS: DOM based XSS is an attack in which the attack payload is executed as a consequence from modifying the DOM environment in the victim's browser. Similarly to reflected XSS, an adversary can trick the victim into using a URL, however, instead of including a script that will target the web application it will alter the DOM environment within the browser. This will result in the client side code contained within the page to be malicious causing it to execute differently due to the malicious content in the DOM environment.

For example, consider a web application that stores a `default` value in a DOM object and that the user interacts with the application via the following URL:

```
http://www.example.com/index.html
        ?default=<some client side value>
```

An attacker can exploit this by having the victim use this URL:

```
http://www.example.com/index.html
        ?default=<script>alert("hacked")</script>
```

This causes the browser to create a DOM object for the page in which the script will be included. The code within the page does not expect parameters to contain scripts within leading to erroneous behaviour.

It is reported that XSS is the second most prevalent issue in OWASP Top 10, being found in approximately two thirds of all applications. There are tools that can actively detect all three kinds of the above-mentioned XSS attacks along with exploitation frameworks. Regarding prevention, the rule of thumb to avoid these vulnerabilities is to always escape, validate and sanitize any potentially untrusted input.

2.1.4 Insufficient Logging and Monitoring

It is intuitive that every system should have proper logging and monitoring so that anomalous activity can be detected and dealt upon. A common and simple example is IP logging in which the system logs the IP addresses from which the users typically connect from in order to detect any anomaly. Lack of monitoring may lead to very severe consequences, such as the 2015 attack on the Ukrainian Power Grid that led to a power outage affecting around 230.000 people from 1 to 6 hours [15]. It is also known that probing from multiple addresses is the precursor of almost every major security incident, allowing for continuous probing increases the probability of a successful attack, and therefore logging can help to understand the level of threat facing an organization.

A possible way to detect lack of logging is by monitoring logs after penetration testing sessions. The actions done by the testers should be sufficiently logged so that an understanding of the damages and how they were caused is possible.

2.2 Static Analysis

Static analysis has the goal of analysing code to help solve implementation bugs. This is of the utmost importance because a simple mistake may lead to a vulnerability that may end up compromising the system or put user data at risk. To achieve this, static analysis examines the code without execution, allowing it to be performed throughout the development cycle, even before having any executable code [31] [21] [37] [48].

Manual auditing can be done over the code, but, it is extremely time consuming and can be almost impracticable in some cases (i.e., applications with millions of lines of code). Static analysis tools are faster and can provide a certain degree of abstraction towards the programmer that executes it, not requiring the operator to have as much expertise as a human auditor. Although it has many qualities, static analysis cannot solve all security related problems. These tools look through the code and find matches with patterns or rules that might indicate a vulnerability. The addition of new rules is crucial for the detection of the vulnerability, but if it is absent the tool with fall prey to false negatives (not alerting for a vulnerability that actually exists). Static analysis tools still require

human evaluation, as there is no precise way to indicate what issues should be solved as soon as possible or which ones pose an acceptable risk level.

Static analysis tools are forced to make approximations, which results in a non-perfect-output. This type of tool can also produce false positives (indicating problems that do not actually exist) in addition to the above mentioned false negatives. Each have their own negative consequences. The false positive may cause the analyst to waste time by looking through code that is completely fine. On the other hand, false negatives give a fake sense of security where no harm shall befall the application when it is, in fact, vulnerable.

Arguably the simplest way to perform static analysis is to utilize the *grep* command searching for particular groups of words. The flaws behind its usage are apparent, since *grep* does not interpret code whatsoever it would not be able to tell the difference between a function call, a comment, or a variable name. Another approach to static analysis is the lexical analysis, which involves preprocessing, tokenizing the source files and matching the resulting tokens with a library of vulnerable constructs. While this approach is one step ahead of a simple grep-based static analysis, it still incurs in many false positives because it lacks the knowledge for the semantics of the code itself.

To improve precision, a static analysis tool may leverage more from compiler technology by using an abstract syntax tree (AST), as it can account for the basic semantics of the program. The scope of the analysis determines the amount of context, as more context reduces false positives at the cost of computation time. Considering the usage of ASTs, the scope of the analysis must be decided among:

- **Local Analysis:** Examine the program one function at a time, with no regard for the interaction between functions.

- **Module-Level Analysis:** Considers interactions within one module, whether the interaction is within functions or class properties, but does not account for connections between modules.

- **Global analysis:** Studies the entire program, and accounts for all relationships among functions.

With regard to Static Analysis several tools have been developed throughout the last decade. In 2006, Jovanovic et al. described Pixy [31] that is a tool for PHP, with the main objective of detecting cross-site scripting vulnerabilities and SQL injections. To achieve the best results possible, Pixy is aware of aliases in PHP, dividing them into two groups: may-aliases and must-aliases. The difference between them is a must-alias will always occur independently of the program path that is taken, and a may-alias is susceptible to conditional statements and is not always guaranteed to happen. To aid the processing, the concept of shadow variables was introduced which, in essence, are local variables introduced by the analysis (cannot be accessed by the programmer), meaning that they cannot

be re-referenced after their initialization. There are two sub-types of shadow variables that represent local variables aliased with a formal parameter and local variables aliased with global variables, respectively. As per evaluation, Pixy has a reasonable performance overhead, but still produces a significant amount of false positives.

In 2008, Wasermann et al. elaborated a static analysis methodology to look for XSS vulnerabilities related to weak or absent input validation [48]. To this end the authors presented an algorithm based on the behaviour of layout engines that check generated HTML documents. The approach taken utilizes string-taint analysis, which starts by translating output statements into assignments to an added output variable, translating the program into a single static assignment (SSA) that encodes data dependencies. This SSA allows for an extended context free grammar (CFG) to be built. The generated grammar is tested against a defined policy, and then checked for untrusted tags which indicate that a vulnerability might be present. This approach has the limitation that it cannot detect DOM-based XSS as this requires an analysis of semantics, and not only the grammar.

Dahse et al. proposed RIPS [21] in 2014. RIPS is a static analysis tool that performs inter and intra procedural data flow analysis by creating backward-oriented taint analysis. By using string analysis the tool is able to validate sanitization in a context-sensitive manner. As per its evaluation, like other tools, RIPS can produce false positives and negatives. The possible causes of the false negatives are field insensitive data flow analysis and second-order vulnerabilities. As per the false positives the main reasons are due to path-insensitive data flow analysis, sanitization through database whitelist and wrong content type.

In 2014, Medeiros et al. presented WAP [37] [28], a static analysis tool that brought new features, namely automatic code correction. WAP not only enables automatic code correction but also uses static/taint analysis to detect vulnerabilities. WAP is separated into three modules, a code analyser, a false positive predictor and a code corrector. The code analyser has the task of examining the code and detecting vulnerability candidates. These candidates have associated to them a tainted symbol table (TST) where each cell is a program statement. The statement allows for collection of data and a tainted execution path tree (TEPT), where through its branches represents a tainted variable and contains one sub-branch for each line of the code in which the variable becomes tainted. The candidate vulnerabilities are then processed in a false positives predictor that uses data mining and machine learning to predict if a given candidate is a false positive or not. After this step, the adjudged false positives are reported as such. Following this, the remaining vulnerabilities that are processed by code corrector, which uses the TST and TEPT to apply fixes, thus eliminating the flaws from the code.

phpSAFE is a static analysis tool for PHP applications proposed by Nunes et al. in 2015 [40]. phpSAFE is required to run on a local web server that has a PHP interpreter enabled and a web browser, and performs static analysis in four stages. First, there is a

configuration stage in which several files are setup in order to specify insecure input vectors, sanitization functions (and functions that can revert sanitization) and functions that might be exploited by an attacker. Secondly, there is a model construction phase in which an AST is generated by splitting the PHP code into tokens. Next, the tokens undergo an analysis stage in which the objective is to follow the flow of tainted variables from input until the output is reached. Finally, occurs the result processing stage in which phpSAFE provides resources that can be used to identify vulnerabilities, aiding with eventual fixes.

PacketGuardian is proposed by Chen et al. [19] as a tool for static detection of packet injection vulnerabilities. PacketGuardian employs precise static taint analysis to check the packet handling logic of various network protocol implementations. PacketGuardian is composed of two main components. First, a taint-based summarizer has the task of crawling through files with a taint style analysis to determine potentially weak accept paths that might lead to packet injection. Next, a vulnerability analyser utilizes the data previously generated and considers attacker-controller data as taint source and packet accept functions as a sink. The output is then analysed to determine if there is either an obvious packet injection or there is a set of protocol states that the implementation relies on to prevent the injection. If the accept paths are protected by protocol states a leakage path constructor performs a second step to find eventual leakage of these states. There is also a partially manual effort in analysis in which a user must identify protocol states.

2.3 Fuzzing

Fuzzing is a testing technique that involves providing erroneous and unexpected data as inputs to a program. The target program is monitored for any strange behaviours such as crashes, memory leaks or failures within the code. The test programs usually take structured inputs and thus can distinguish valid from invalid inputs. The goal of the fuzzer is to generate inputs that are valid enough to bypass the parser but invalid enough to cause erroneous behaviour within the program, exposing any possible bug [10].

There are three main criteria that can categorize a Fuzzer:

1. How it generates input
2. How aware it is of the input structure
3. How aware it is of the program structure

Regarding input generation, a fuzzer can create inputs by resorting to two different approaches, generation and mutation. A generation-based fuzzer randomly builds inputs without any information and is not dependent on the existence of seed inputs. On the other hand, a mutation-based fuzzer leverages from existing seeds by creating and mutating this information to produce new input values [27].

The application code distinguishes valid input that is accepted and processed by the program and input that can quickly be rejected. The factors that determine a valid input can be explicitly specified in the input model. It is worth noting that items that are normally not considered as input can still be fuzzed, like shared memory and environment variables. In this context, a smart fuzzer leverages from the input model to generate more valid inputs. If the input can be modelled as an AST, mutations could be done over random transformations to complete subtrees from node to node. In the case that the input is modelled by a formal grammar, a smart generation-based fuzzer could produce input based on the rules of the grammar. Obtaining this input model might be difficult due to proprietary issues, complexity or simply because they are unknown. A dumb fuzzer, on the other hand, does not need an input model and can be used to fuzz a broader range of application. The cost is having less valid inputs and being more demanding on the program parser rather than the main components that implement the application functionality.

As per the awareness of the program structure, a fuzzer can be one of the three following types:

- **Black-box:** Treats the program as a black box by remaining completely oblivious to the internal structure. Although this allows for fuzzing of many distinct programs, due to lack of modelling, a blackbox fuzzer usually only scratches the surface and fails to discover more intricate and complex bugs [9] [24] [23].

- **White-box:** Takes advantage of a program analysis to increase code coverage and attempt to reach critical program locations. Contrarily to a black-box fuzzer, it can be very effective at exposing bugs hidden deep within the program. However, the time required for the analysis can become impracticable as it takes rather long to generate the inputs.

- **Grey-Box:** A grey-box fuzzer attempts to leverage from tracing as opposed to program analysis to infer information about the code. This does cause a significant performance overhead but the aditional knowledge grants increased code coverage during fuzz testing, making grey-box tools an efficient vulnerability detection approach that attempt to extort the best features from both black-box and white-box fuzzing [11].

Next, we will review a few example fuzzers. In 2010 Bau et al. presented a state of the art for automated black box fuzzing for Web Applications [10]. The main objective of this study was to check what vulnerabilities were mostly tested by the scanners, if these vulnerabilities were representative of a real world scenario and the effectiveness of these scanners. The study indicated that the most extensively tested vulnerabilities were Information Disclosure, XSS, SQLI and Cross Channel Scripting. The main conclusions drawn from this analysis is that there was no perfect scanner that excelled in all

types of vulnerabilities. This leads to believe that collaboration between developers by exchanging ideas would help to develop a better black box scanner. Generally, scanners do test a significant proportion of real life vulnerability scenarios. Finally, the vulnerability detection rate was, globally, lower than 50%. This however does not compromise the usefulness of black box testing, and in practice it is a prevalent methodology for detecting vulnerabilities.

Antunes et al. describe AJECT [9], a tool that aims to find vulnerabilities by injecting attacks similarly to a hacker or security analyst in network-connected servers. AJECT treats the server as a black-box, but, it requires a communication protocol specification with the server to generate intelligent attacks. The protocol specification could be provided via a GUI. An Attack Injector then processes the produced attacks and then injects the necessary packets to the server. The server's response is collected and matched with a server-side monitor that inspects execution, allowing for a more accurate identification of the unexpected behaviour that was caused. AJECT was tested against 16 fully patched and up-to-date servers and discovered vulnerabilities in 5 of the tested servers.

In 2012 Holler et al. proposed LangFuzz [27], which is able to exploit project-specific knowledge while relying solely on generic fuzz testing. The way this is achieved is by employing primarily mutation but also generation. This way semantic rules can be respected by reusing existing semantic context while staying language independent. One of the drawbacks is that by not having semantic background there is a chance of obtaining insane programs. When stacked up against jsfunfuzz [7] (a state of the art fuzzer for javascript), LangFuzz detected 8 defects where jsfunfuzz detected 15. However, there is only a slight overlap on the identified bugs meaning that LangFuzz was able to detect different kinds of defects, thus still providing a significant contribute.

LigRE, proposed by Duchéene et al. [24], takes a reverse-engineering approach that guides fuzzing towards detecting XSS vulnerabilities. It starts by learning the control flow of the application by means of a state aware crawler. The inferred model is then annotated with data flows of input values into outputs, thus producing a control plus data flow model. After the model is constructed, the most promising annotations are prioritized while slicing and fuzzing. The slices are pruned models that permit to drive the application to a given state by sending a certain malicious value, and then guide the fuzzer to navigate towards the destination and observe the effects of the processing of the malicious value.

KameleonFuzz [23] is an extension to LigRE that employs evolutionary fuzzing, thus generating input to exploit XSS and detecting how close it is to actually revealing a vulnerability. KameleonFuzz replaces the fuzzing portion of LigRE with two new steps, Malicious Input Generation and Precise Taint Flow Inference. A genetic algorithm is parametrized by an attack grammar and evolves malicious inputs. Additionally, a fitness function is also used so that most suitable inputs are favoured and evolved for attacks,

treating each input sequence as an individual and weighting it accordingly based on its proximity to exploitation. Regarding evaluation, KameleonFuzz showed no false positives and was able to surpass the other fuzzers. The limitations from this approach are shared with LigRE, with the additional issue of requiring a manually built attack grammar, and it relies on the hypothesis that XSS is the result of only one fuzzed value. Hence, vulnerabilities that might involve two or more fuzzed values simultaneously will not be discovered.

In 2017 Bohme et al. presented AFLGo [11], a directed greybox fuzzer that aims to generate input that can efficiently reach a set of target program locations. Thus it serves as a useful patch testing tool, as it can be directed towards recently introduced vulnerabilities or incomplete fixes. AFLGo leverages from metrics that indicate the distance to the target functions in the call graph, along with Markov chain modelling, and an annealing-based power schedule. Function-level target distance determines the distance from a function to all target functions in the call graph. Basic-block-level target distance corresponds to the distance between a basic block to all other basic blocks that call a specific function. Both these distances are used to calculate the normalized seed distance. The seeds are subject to an annealing-based power schedule that assigns more energy to a seed that is closer to the targets, while also deterring the energy from seeds that are further away. The simulated annealing algorithm converges towards the set of seeds that exercise the maximum target locations (generically they converge towards optimal solutions). It relies on two parameters, the temperature that regulates the acceptance of worse solutions and the cooling schedule that controls the rate of convergence of the algorithm. Regarding execution, AFLGo initially extracts call graphs and control flow graphs from the source code. Next a distance calculator finds out the basic-block-level distance for each basic block. Thus it is then used by an intrumentor that generates a binary that ultimately informs the fuzzer about both coverage and seed distance. This information is used while fuzzing to produce subsequent inputs. The approach proved to be effective, as AFLGo was able to outperform several vulnerability detection tools. Furthermore, it also proved capable of discovering vulnerabilities in error-prone software components that are often patched.

2.4 Symbolic Execution

Symbolic execution in software testing has the goal of exploring as many different program paths as possible in a certain amount of time, while generating, for each path, a set of concrete input values that exercise the said path and checking for the presence of various errors, including security vulnerabilities. The main idea behind symbolic execution is to replace actual input data with symbolic values, and represent values assigned to the program variables as symbolic expressions [14] [13].

Christensen et al. describe a symbolic analysis approach to map Java programs into finite-state automata [20]. An approach utilizing flow graphs is used to capture the flow of strings and the string operations in a program while abstracting everything else away. Nodes correspond to variables or expressions and the edges represent possible data flow. From this flow graph a special context-free grammar is built and approximated into a regular grammar using a variant of the Mohri-Nederhof algorithm. Finally this grammar is transformed into a Multi-Level Finite Automata (MLFA) from which several Deterministic Finite-state Automatons (DFA) are extracted.

In 2008, Cadar et al. presented KLEE [12], which is arguably one of the most popular symbolic execution tools. KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter. Its core is an interpreter loop that selects a state to run and then symbolically executes a single instruction in the context of that state. This loop runs until there are no states left or a predefined timeout is met. Registers, stack and heap objects refer to expressions, represented as trees, instead of raw data. The leaves of an expression are symbolic variables or constraints and the interior nodes come from the assembly language operations.

Later, in 2011, Cadar et al. [13] explained several advances to symbolic execution, which happened mainly due to algorithmic improvements and the increase in available computational power. The paper details some tools that have been developed more recently while describing two different methods of symbolic execution. The first is Generalized Symbolic Execution (GSE), which has the ability of handling multi-threading and program fragments with recursive data structure inputs. GSE leverages from an off-the-shelf model checker, whose built-in capabilities support multi-threading. GSE handles input recursive data structures via lazy initialization. It starts executing the method on inputs with unitialized fields, and it initializes the fields non-deterministically when they are first accessed during the symbolic execution. This allows for the symbolic execution of program methods without requiring an a priori bound on the number of input objects. The second method is Dynamic Test Generation (DTG), which improves traditional symbolic execution by distinguishing between the concrete and the symbolic state of a program. The code is run without modifications, and only the statements that are dependant on the symbolic input are treated differently by adding constraints to the current path conditions.

In 2012 Marinescu et al. proposed ZESTI [35], a technique that amplifies the effect of regression testing. It checks the program paths explored by the regression test suite against all possible input values and it explores additional paths that diverge slightly from the original execution paths to search for potentially dangerous operations. ZESTI is an extension of KLEE that improves the testing of sensitive operations, explores additional paths around sensitive operations and enhances efficiency by discarding less relevant test cases. These three improvements are important in the context of security as it is more desirable to explore sensitive sinks in the code than other parts. ZESTI achieves this by

dedicating more resources to test critical operations. For example, the fact that ZESTI discards some test cases allows the execution to terminate in some cases that KLEE does not.

In 2013 Cadar et al. presented another work [14] that documented the evolution of symbolic execution over the past 30 years, and detailed it some of the more modern approaches taken. Among these approaches is Concolic Testing or Directed Automated Random Testing (DART). Concolic testing dynamically executes the program also with concrete inputs. This way concolic testing keeps both a concrete state that maps all variables to their concrete values and a symbolic state that only maps variables that have non-concrete values. Concolic execution runs the program with some given or random input, gathers symbolic constraints at conditional statements along the execution, and then utilizes a constraint solver to infer variants of the previous inputs. This is used to steer the next execution of the program towards a particular path. The process is then repeated systematically or heuristically until either the time budget expires, all execution paths are explored or a user defined coverage criteria is met. Another recent approache is Execution-Generated Testing (EGT). It intermixes concrete and symbolic execution by dynamically checking before every operation if the involved values are all concrete, and in that case it executes just like a normal program. Otherwise, if one or more values are symbolic, the operation is performed symbolically, by updating the condition for the current path.

Some of the challenges posed by symbolic execution are also described. One of them is path explosion, which results from the exponential number of branches in the code. This makes it critical to explore the most relevant paths first. A possible solution to this problem is heuristic techniques that can focus the symbolic execution in parts of the program that are considered most relevant. Another possible solution is to use sound program analysis techniques which aimed to use diverse ideas from program analysis and software verification to reduce the complexity exploration in a sound way. Constraint solving is one of the key bottlenecks of symbolic execution that dominates runtime and is one of the main reasons why symbolic testing fails to scale with some programs. The two proposed solutions are eliminating irrelevant constraints by, for example, removing constraints that do not decide the outcome of the current branch. An alternative approach is incremental solving. This approach aims to take advantage of the fact that similar constraints lead to similar solutions. For example, removing a constraint does not invalidate a previous solution that considered more constraints.

Dowser was presented in 2013 by Haller et al.[26]. It is a guided fuzzer that combines taint tracking, program analysis and symbolic execution, to find buffer overflow and underflow vulnerabilities. At a high level Dowser starts by using static analysis to find array accesses in loops and ranks them based on a function that prioritizes more complex operations. Next, Dowser picks high-ranking accesses and selects test inputs that exercise

them. Then, dynamic taint analysis is used to determine which input bytes influence the pointers dereferenced in the candidate instructions. The objective is to have Dowser test only the field that affects the vulnerable memory access candidate. For each candidate instruction and input bytes involved in calculating the array pointer, Dowser uses symbolic execution to try and cause a buffer overflow. Lastly, to detect if any overflow occurred, Google's AddressSanitizer is used.

In 2012, Jensen et al. presented THAPS [30]. THAPS is a tool that is based upon the symbolic execution of PHP, focussing primarily on SQL injection and XSS vulnerabilities. THAPS executes taint analysis based on the symbolic execution that identifies the user input and how it propagates through the application. If the tainted data reaches any sensitive sink without being properly sanitized, it reports a vulnerability. Symbolic execution is done by traversing an AST generated by the PHP-Parser library. For the simulation of all possible outcomes, the analysis might need to store several values for the same variable due to multiple assignments inside different code branches. These values are saved in a variable storage that also records what branch of the code the value belongs to. The analysis itself is done in 5 steps: Body Analysis, where the global scope and the bodies of methods are analysed; Pattern Detection, where certain idioms or programming patters that need special handling are recognized and processed; Class Processing, where classes are analysed resulting in a description of the class members and methods that are used when running body analysis; Function Processing, where functions are analysed and a description of them is stored; and lastly, Inclusion Resolving, where the AST nodes that represent a file inclusion are replaced with the AST generated from the included file. THAPS implements the notion of a framework model which is a template that superficially implements the core API and functionality from the application framework. This way THAPS can obtain some more meaningful context. THAPS also employs dynamic analysis to overcome some uncertainties of static analysis. Dynamic analysis is done through an extension for the Zend Engine that converts the parsed PHP code and interprets it. The extension connects a database and generates a unique identifier for each analysis run. Once the dynamic analysis is done, static analysis is performed with the additional information. To automate the process of making a request to a page, THAPS utilizes a version of Crawler4j that was modified to support access to the special cookies.

Driller, described by Stephens et al. [44], is a hybrid vulnerability discovery tool that leverages fuzzing and concolic execution to find bugs deep within the code. These techniques are used in a complementary manner in which fuzzing exercises code blocks of an application and concolic execution is employed to generate inputs that satisfy complex checks that separate blocks. Driller operates through multiple components. Driller starts by using its fuzzing engine that explores the first block of the application until it reaches a complex check on a specific input. At this point the concolic execution component is invoked to "unstuck" the fuzzing engine. Concolic execution considers the pre-constraining

user input with the unique inputs discovered by the previous fuzzing step to avoid path explosion. After tracing the inputs, the concolic execution uses its constraint solving engine to determine inputs that would force execution through previously unexplored paths. Once the new input is identified, the fuzzing engine continues to mutate the inputs to test new blocks.

Chainsaw [8] is a tool for web vulnerability identification and exploit generation that builds injection exploits that span several HTTP requests. The first step of chainsaw is to create a seed that considers each module in isolation and builds a model of the computation on its inputs. In concrete terms, this approach gets exploit seeds that are a verified set of inputs that direct the execution to a vulnerable sink inside a model. To obtain these seeds, for every sink inside a module, Chainsaw explores all execution paths from sources to that sink symbolically and for each sink computes a symbolic sink expression. Then, Chainsaw gets a navigation graph called General Workflow Graph (GWFG) that represents all possible navigation sequences and derives a ranking for those sequences. For every global execution path that leads to an exploit sink, Chainsaw creates a symbolic formula that represents the computations and conditions along that global path. Lastly, a solver attempts to find a solution, which contains values to all local and superglobal variables that need to be provided to every module in the navigation sequence in order to reach the exploit seed. Another of the advantages of Chainsaw is the ability to model the database state, thus enabling the discovery of first or second-order exploits. To this end, Chainsaw uses two distinct techniques: Static Input Generation builds a mapping between writes to a database and the queries that read the same table; Dynamic Auditing, in which Chainsaw uses existing database data of a live system by including constraints arising from its state in the symbolic execution.

2.5 Oracles

An Oracle is a module that can be used to detect attacks at runtime. Although it may come as server side modules [36], the most common form of oracles as proxy servers. A proxy can be placed between a service and a client in order to monitor what kind of traffic is flowing through their communication channels. This is extremely useful as monitoring is a viable way to extract Indicators of Attack (IoA), thus allowing for proactive measures to be taken before any severe damages occur (e.g., packets with malicious contents and blacklisting users that attempt malicious actions). Oracles can range from simple, just perform a labelling action as an attack or non-attack, to more complex, thus providing additional context about the action that is being attempted.

SQLProb was presented by Liu et al. [33] with the intention of detecting SQL Injections. The approach places a proxy between the web application and the database. The proxy is composed of four components: A Query Collector that processes all SQL

queries during a data collection phase; a User Input Extractor that identifies user input data; a Parse Tree Generator that creates the parse tree for incoming requests; and finally a User Input Validator that evaluates if the user input is malicious or not. The approach causes a higher response time due to query processing but CPU usage overhead is fairly low.

FlowWatcher [38] is an HTTP proxy that aims at mitigating data disclosure vulnerabilities by monitoring HTTP traffic and by partially shadowing the application's control state based on a User Data Access (UDA) policy. UDA policies allow for developers to specify an intended access control model for the application by relating users to the data that they are allowed to access. One of the limitations of these UDA policies is that they rely on specific URL and HTTP request and response formats. In this case, a change in URL would require a change in the policies as well. FlowWatcher is implemented as a Nginx proxy server placed between the client and the web application. Whenever a client sends an HTTP POST request, it is intercepted and matched with the UDA policy. The request is then forwarded to the application. FlowWatcher intercepts the response consulting the policy to confirm that the response does not contain data that should not be seen by the user who made the request. If the user has clearance for the information then the response is sent without any modification, otherwise, the response is filtered such that the data items that user should not be able to see are removed.

To prevent SQL Injections without using a proxy, Rauti et al. proposed a SQL diversification technique [43]. The solution is composed of two layers, a set of diversified SQL symbols that depend on specific applications and queries, which the SQL server will decode when they arrive. In practice each application would have to encode their SQL queries, and the SQL server would then process these queries with a decoder and execute them. The approach introduces little to no performance overhead as it needs no proxy and only requires string replacement. Some of the limitations of this approach are that all applications on the server need to be diversified and direct access to the SQL server's source code is needed, although the changes are reportedly minimum.

George et al. proposed an architecture for query anomaly detection and SQL Injection prevention [25]. This architecture that should work in three phases. The first phase involves classifying a set of SQL injection queries based on both their sub-types and technique used for injection. Every incoming query is then checked and, in case of anomaly, the query is reconstructed according to the standard query template before being forwarded to the database. In the second phase, techniques for detection and mitigation of SQL Injection are implemented. Automatic training methods are used to process the defined SQL constructs. Finally, the SQL Injections are checked with the vulnerability detection system to determine the total number of queries forwarded.

Proposed in 2016, SOFIA [18] is a security oracle that is independent from known instances of successful attacks. It has no knowledge of what input data is used to test

the system and does not rely on the source code of the web application. To achieve this, SOFIA consists of two phases a training and a testing phase. Training data is provided in order to construct a model of safe executions, thus classifying all remaining scenarios as anomalous. Next, five steps are then executed. First, there is a parsing step in which SQL statements are parsed into trees to detect SQLI attacks. Next, the statements are pruned, which means that all constant numeric and string values in the parse tree are replaced by a place-holder. This is done as legitimate queries should differ only in value when compared to malicious queries that include code. The third, and last shared step between training and testing phases, is computing distance, in which tree distances are calculated among queries to determine their similarity. However, this is not enough to determine if a SQL query is an attack and there is still the need to undergo a clustering step. Clustering uses the k-medoids algorithm, where representative samples are used to form clusters and minimize the dissimilarities. Finally, the classification step evaluates the query based on its diameter and distance to the nearest cluster.

SEPTIC is a mechanism proposed by Medeiros et al. [36] to prevent DBMS injection attacks, which can also assist in vulnerability identification. SEPTIC works as a module inside the DBMS. Whenever a DBMS receives a query, it parses and validates it. Then, before its execution SEPTIC, is called to detect and block potential attacks. Whenever a new application is used SEPTIC needs to go through training where it is required to execute all queries of the application with valid inputs in order to create models of the queries. SEPTIC also does vulnerability diagnosis and removal by composing the produced models at runtime with the ones previously stored.

Dalai et al. recently proposed a method to neutralize SQL Injections within web applications [22]. This approach is similar to SEPTIC, but however instead of checking the query inside the DBMS it is treated instead inside the web application server. The approach itself is composed of six steps. First, when a SELECT query is received, all characters after the WHERE clause are extracted and stored. Then, the input parameters accepted from the user are checked for their appropriate type. Next, the query string is normalized and converted into a simple statement, replacing any encoding if need. After this, a string extraction method is used to get all the characters after the WHERE clause. The input parameters from the extracted string are then removed sequentially. The resulting string from the first step and the last step are then compared. If they match, the query is safely sent to the DBMS for execution. If not, the query is dropped and the user is informed with a warning indicating that a SQL Injection is being attempted.

2.6 Indicators of Compromise and Indicators of Attack

Indicators of Compromise (IoC) and Indicators of Attack (IoA) are extremely relevant in the context of computer forensics, although both are useful when it comes to detecting

threats within a system. There are some significant differences between both of these artefacts which we further detail.

2.6.1 Indicator of Compromise

An IoC is an artefact that can be observed on a system that indicates with a high likelihood that an intrusion might have taken place in the system.

Traditional IoCs are virus signatures, URLs or domains associated with botnets, hashes of malware files or IP addresses associated with malicious activity. An IoC can be considered more of a reactive measure than proactive. Indicators are usually exchanged within the industry in order to generate a broader awareness of what might denote a compromised system.

In 2012 Lock from the described IoC usage in malware forensics [34]. More concretely, it sheds some details on the OpenIOC framework that enables the description of IoCs on a XML based syntax and allows for easy conversion to intrusion detection tools such as yara [45] or snort [16]. Although OpenIOC does provide a simple and effective way of describing malware infections, thus far it can only describe low-level, host and network attributes. Hence, it lacks the syntax to provide semantic consideration of the attributes.

iACE, presented by Liao et al. [32], is a tool for fully automatic IoC extraction. iACE's architecture is composed of a Blog Scraper (BS), a Blog Preprocessor (BP), a Relevant Content Picker (RCP), a Relation Checker (RC) and a IoC Generator. The BS automatically collects articles from blogs while removing irrelevant information. The articles are then processed in the BP with Natural Language Processing (NLP) to remove any article unlikely to contain a IoC. For each article that is considered relevant, the RCP converts all its content to text and breaks it into sentences and other special content elements. Then, it searches among the sentences for those that might involve IoCs with a set of context terms and regexes, that describe the formats of the IoCs. For each sentence, the RC analyses its grammatical structure connecting the context terms and the IoC anchors, using a learned model to determine whether the latter is in fact an IoC, and if so, mark the IOC and context terms. Using this labelled information the IG then automatically creates the header and the definition components, including indicator items for all IoCs identified, following the OpenIOC standard.

In 2016 Catakoglu et al. proposed an approach that allowed for automatic extraction and validation of IoCs for web applications [17]. The solution consists of two main components. A high-interaction honeypot comprised of multiple VMs that are tasked with collecting a large quantity of pages compromised by attackers. Each VM is responsible for running a different web application isolated in a Linux container. Attackers exploit these applications they can modify existing web pages and install new ones. Each VM automatically collects and compares the original state of the container with the exploited

state. Once a difference is detected, all the modified and uploaded files are extracted by the system. The second component is in charge of preprocessing the obtained data and automatically extracting URLs of the components that were remotely included in the attackers files, then storing them along other relevant information in a database. Along with the URL, dates in which the the URL was first and last seen on the honeypot are stored. URLs are probed daily to verify it they still return a valid response, in case there are no errors the ”last good response date” variable in the database is updated. Lastly, a counter of how many times a URL is included in uploaded components is also kept.

2.6.2 Indicator of Attack

Whereas an IoC attempts to identify a compromised system, the objective of an IoA is to detect what the adversary is trying to accomplish independently of malware or the exploit used in the attack. This means that even though an attack might be under way, it has not necessarily been effective at compromising the system.

An IoA represents a sequence of actions that an adversary must take to make his attack successful. A good example for this is spear phishing. Spear phishing involves several steps: persuading the user to open a link providing a gateway to infect his machine; discretely executing other processes; hide in disk and secure persistence through system reboots. An IoA, in this case, would be concerned with the execution of said steps, and the goal behind the attack. A document was published by Intel Security providing a description of IoAs [6], where the usage of this technique is encouraged as it allows for earlier action against threats. Important considerations are to collect the data and not just observe and discard it, enrich with contextual data and analyse with some human factor experience. Centralized services help in both retrieving and contextualizing the data as correlations become easier to make. This way, when an IoA is created, people and processes can act while the intelligence is distributed by emitting alerts or enforcing quarantines in nearly real time.

Content Security Policies (CSP) have been standardized by W3C and adopted by all major browsers. CSP is an approach to avoid XSS attacks, however, even though the client-side has been successful the same cannot be said about server-side, with a very minute number of sites having actually enabled CSP. Pan et al. propose CSPAutoGen [42], a solution that automatically generates CSP based on templates, rewrites pages in real time to apply the generated CSPs and passes the pages to the browsers. CSPAutoGen has three main phases. In the initial training phase, a URL list is identified. These URLs are then used by a rendering task generator to create tasks by specifying various cookies. During rendering, all relevant content is sent to a template generator and a host whitelist generator. The resulting templates and host whitelists are stored in a database and can be updated if needed. Next, during the rewriting, CSPAutoGen extracts and sends inline scripts to a JavaScript server that matches the scripts against the templates, and stores the

benign scripts in a trusted subdomain. An applier engine then rewrites the webpage to import the trusted inline scripts as external scripts and includes the domain's templates. Lastly, during runtime, CSPAutoGen uses a client-side library to handle runtime included inline and dynamic scripts. For inline scripts, once the client-side library detects DOM tree changes and inline scripts inserted, the library matches the scripts with the templates. If these match they are sent to the JavaScript server that matches them once more and adds stores them in case they are benign. Next, the URLs associated with these scripts are returned so that the client side can load and execute them. In the case of dynamic scripts, the JavaScript library detects any calls to eval-like functions and matches their parameters with the templates. If matched, the library evaluates the scripts by instantiating corresponding symbolic templates.

Chapter 3

Leveraging from Monitoring for Vulnerability Discovery and Removal

This chapter looks into our proposal for enhancing the security of web applications by leveraging from the outputs of monitoring. We provide an overview of the architecture of our monitoring approach, and the technologies it uses, along with possible variations. Furthermore, we expose the benefits of this approach along with different deployment options as the architecture can either be fully or partially implemented, depending on the needs of each system and message traffic that is monitored. We also introduce one of the key techniques, the *directed static taint analysis*. Section 3.1 provides a general view of the architecture. Section 3.2 offers a more detailed look into each module that composes the architecture.

3.1 Architecture

This section presents the proposed architecture along with the key concepts and modules it relies on. The architecture enables the usage of various types of information to detect vulnerabilities with the ability to cope with possible false positives and/or false negatives. We utilize five modules to extract and process information, which is used to fine tune the analysis to taint only certain fields as well as to patch vulnerabilities that are found in the analysis process. We believe that this architecture is versatile in the sense that it does not require a complete implementation of all modules and can be the target of alternative configurations depending on the scenario or choices of the developers. Furthermore, we introduce the concept of directed static taint analysis, which we utilize to make the processing of the application's code more efficient and precise.

The main architecture is presented in Figure 3.1. The modules of the architecture are:

- *Input Injector:* allows for data (input) to be sent to the application. This can be done manually by a client (e.g., a browser) or through a *fuzzer*;
- *Input Interceptor:* captures the injected inputs in the network and determines if they

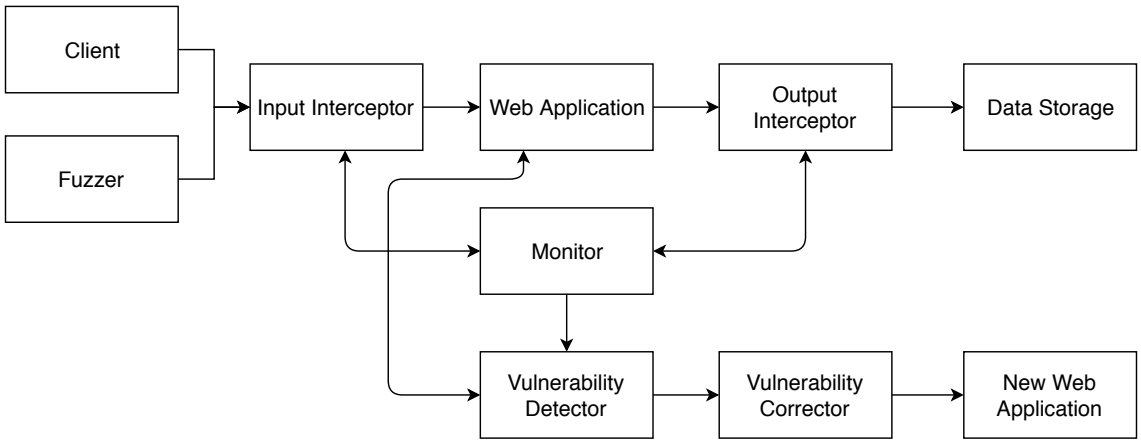

Figure 3.1: Proposed architecture.

are malicious by analysing their contents based on a set of rules. Depending on how the configuration is carried out, it may always allow for the input to flow to the web application or it may let the input pass only when it is considered benign;

- *Web Application:* the web application that receives the inputs sent by the user and executes some service;
- *Data Storage:* a place to save data, such as a database and/or a file system, which the application might interact with to save and retrieve information;
- *Monitor:* oversees and coordinates the actions between vulnerability discovery and the intercepted input and/or output, and registers all ongoing activity;
- *Output Interceptor:* intercepts the output of the web application before it reaches the data storage. It analyses the output and drops the traffic if there is any malicious content and reports the result back to the monitor;
- *Vulnerability Detector:* executes a directed static taint analysis of the web application code, based on the collected malicious data and the reports produced by the input and/or output interceptor. Identifies the necessary information to remove the vulnerabilities and confirms the existence of a true or false positive in case the output interceptor is not being used;
- *Vulnerability Corrector:* automatically corrects the vulnerabilities found in the code by using the results of the analysis done by the vulnerability detector. It produces a new safe version of the web application, which is ready to substitute the one that is currently in execution.

With all the modules implemented there are two forms of operation in which a vulnerability can be detected:

- Output Interceptor - If a malicious input flows through the code of the web application and reaches a sensitive sink, it will result in a malicious request to the data storage (e.g., a malicious query). Therefore, by correlating the input and the outputs of the web application it is possible to accurately identify the request as an attack, which confirms the existence of an exploitable vulnerability.

- Vulnerability Detector - By statically analysing the code via the attack surface that the received input would explore, one can determine if it would reach any of the sensitive sinks of the application.

Note that the output interceptor requires a successful attack to be able to determine if a vulnerability is present. On the other hand, if the vulnerability detector is provided with the correct attack surface during parametrization, it will detect the vulnerability regardless of a successful attack or not. The immediate benefits of this approach are twofold. First, by having a directed static analysis, we get a higher efficiency while still maintaining the accuracy of static analysis. Second, the analysis will only be triggered if there is indeed a possible exploitation as proven by the output interceptor module. Otherwise, the system will keep operating without any analysis being done, thus saving time and resources.

Based on the alternative the ways of discovering vulnerabilities, the architecture enables the following approaches:

1. *Guided static analysis based on warnings produced by the Input Interceptor* - In this case the output interceptor module does not require implementation and the static analysis is based solely on the information that is provided by the input interceptor. With this implementation, we avoid having any component between the application and the storage system, thus eliminating some of the overheads. In any case, it is still possible to direct static analysis by using warnings produced by the input interceptor and extracting the parameters in malicious requests. This possibility is illustrated in Figure 3.2.

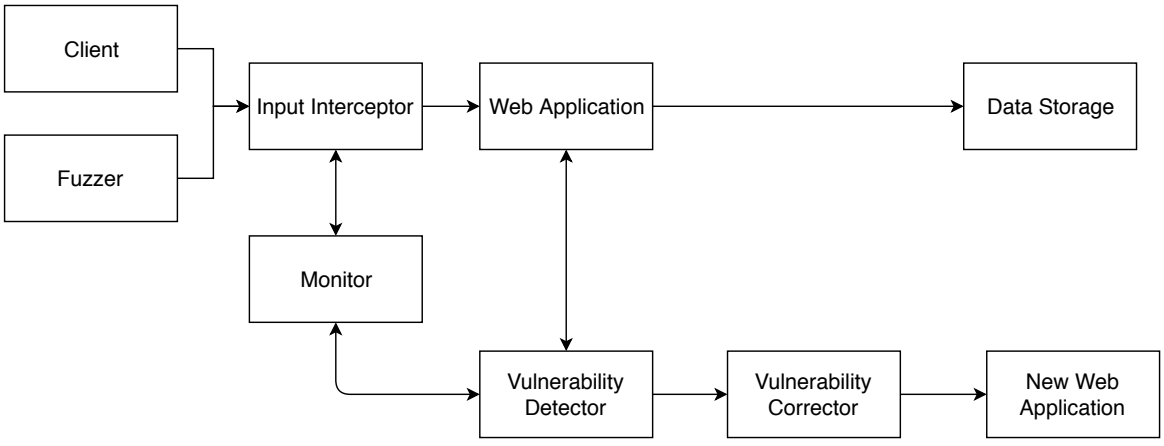

Figure 3.2: Proposed architecture focusing on the input interceptor.

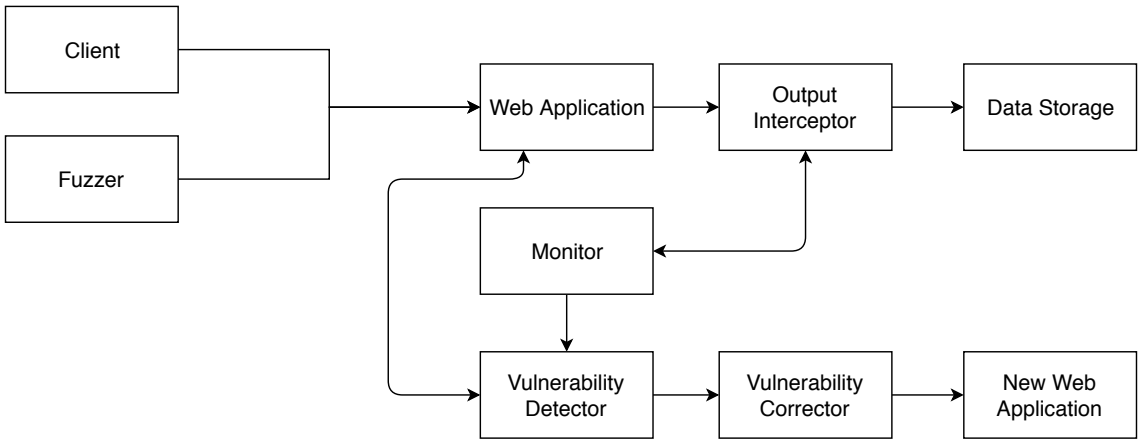

Figure 3.3: Proposed architecture focusing on the output interceptor.

2. *Guided static analysis based on malicious requests detected by the Output Interceptor* - The static analysis is based solely on the information from the output interceptor, and therefore the input interceptor module does not require implementation. Since this approach is not dependent on input interceptor it may become harder to extract the parameters that have reached the program. However, the analysis will only be done when the intercepted output is malicious, thus reducing the number of times the analysis is made. This approach is represented in Figure 3.3

3. *Guided static analysis based on warnings produced by the Input Interceptor and malicious requests detected by the Output Interceptor* - Static analysis can be guided via both input and output data. This allows for a more refined code analysis along with a reduction in false negatives and false positives. This scenario is illustrated by Figure 3.1

3.2 Main Modules

This section will provide a more detailed description of the components that form the architecture.

3.2.1 Client & Fuzzer

The client is responsible for providing the inputs that will reach the web application. We also contemplate the utilization of a fuzzer to provide malicious inputs to the rest of the architecture and trigger the functionality of the remaining components.

3.2.2 Input Interceptor

The input interceptor module captures the requests that flow from the client (or fuzzer) to the web application. A request is composed by the URL, the parameters and the corresponding values. Generally, the structure of a request is as follows: `<url>?<param>=<value>&...& <param>=<value>`, where `<url>` represents the base URL of the web application and may contain the name of the file that is targeted by the input; `<param>` a parameter; and `<value>` the data assigned to the parameter. The parameters are the entry points of the application and are defined within the source code, which for example, can receive the values filled in an HTML form.

This component can be viewed as a proxy that receives all requests sent to the application and it has the capacity to: (1) determine if the data transmitted is benign or erroneous by employing a set of rules that identify the patterns from malicious activity. For example, the usage of `'` followed by an SQL statement represents an SQL Injection and the values that contain `<script>` usually are employed in XSS attacks; (2) generate alerts in the case that malicious activity is identified. Every time that the data contained in a request matches one or more of the defined rules an alert is produced containing some information about the problem.

In more detail, the alert is composed by one or more messages that identify the malicious activity and the request that is being analysed, which can be considered as a type of indicator of attack, as it will be created by finding known malignant patterns within the inputs that with a high likelihood represent an attack. A generic alert representation is:

```
Attack 1 : <description of the attack>
Attack 2 : <description of the attack>
...
URL: <The URL that generated the warnings>
```

The alert is sent to the monitor so that it can keep a log of the activity and trigger the vulnerability detector to start the static analysis depending on the verdict provided by the output interceptor (if the third approach is chosen). If there are no signs of erroneous activity, the request is classified as benign and it will be forwarded to the web application, proceeding with the normal execution.

Note that the input interceptor does not have any information about the code, and therefore false positives might exist. Recall that even if an input is indeed an attack this does not necessarily mean that a vulnerability exists. The same can be said for false negatives, as a warning might not be issued even though there is an attack present.

It is important to choose a tool for the input interceptor and configure it so that it suits the needs of the system accordingly and balances false positives and false negatives. However, to provide further assistance with reducing false negatives and positives, we employ an output interceptor module that analyses the data that flows from the web application to

the data storage (see next section).

Even when a malicious input is provided, it might not be enough to exploit a vulnerability as the bug may have already been patched. Hence, every time an input triggers a warning, the need for the analysis of the code itself is still dependent on the output interceptor module and on the past activities of the vulnerability corrector. After this matching is done, the warnings and a copy of the packet are sent to the monitor. This is done to limit the considered attack surface for each execution of the static analysis.

3.2.3 Output Interceptor

This module is positioned between the web application and the data storage, such as a database or the file system. It has the primary function of intercepting and analysing the data storage request in order to determine if it corresponds to an attack or not. In case of an attack, the output interceptor module will contact the monitor with confirmation of an attempted exploit. It will also drop the request before it reaches the data storage, thus preventing the attack from taking place. This allows for a more accurate labelling of both true positives and true negatives, as ideally only when there is evidence that an attempted attack has taken place will the monitor order a directed static analysis and subsequent vulnerability correction.

The output interceptor is the most trustworthy point of the architecture for the detection of attacks as it analyses the traffic that results from the execution of the web application. However it can still make mistakes, raising alarms unnecessarily or letting attacks through. If the input interceptor does not raise an alert initially but the output interceptor detects an attack, we can draw the conclusion that probably the input interceptor has issued a false negative. On the other hand, if the input interceptor detects and issues an alert which later the output interceptor declares to be harmless, it can be inferred that the first alert was likely a false positive.

The reasoning behind this approach is to check for IoAs in the data request that is intercepted between the web application and the data storage, such as attempts of execution of arbitrary commands or code injection. In terms of the representation of the alert that is transmitted to the monitor, it has a similar format to the alerts created by the input interceptor:

```
Attack 1 : <description of the attack>
Attack 2 : <description of the attack>
...
Request: <The request sent to the data storage that was
 intercepted and generated the alarms>
```

To identify malicious queries, we can leverage from existing oracle technology like SEPTIC [36]. This kind of approach constructs a model of the query, which is then used to run

verifications over the observed request. This model represents the query and allows the identification of query elements and input types. The usage of this model also supports the validation of the structure of the query. For example, if the model and query are different regarding the number of items, we can assume that their structure does not match and this usually can only occur if an attack is underway. It is also possible to syntactically verify the query by checking that every element or data type matches those of the model. Lastly there is also the possibility of analysing queries by similarity. This is done by checking if the string elements in the initial query are distinct from the those in the query structure.

3.2.4 Monitor

The monitor module is directly connected to the vulnerability detector, input interceptor and output interceptor. It has the goal of managing the information received from both ends and then trigger the vulnerability detection. It is responsible for the following main actions: (1) management of the alerts generated by the input interceptor and output interceptor; (2) configure and trigger code analysis from the vulnerability detector.

Alert Management

The alerts sent from the input interceptor are analysed by the monitor to determine if they correspond to attacks that try to exploit vulnerabilities. There is the chance that the alert is not related to a vulnerability but to abnormal traffic (e.g., using IP addresses as input in the URL). However, the information received from the output interceptor is also relevant as it may report an ongoing attack that was misjudged by the input interceptor, leading to four possible scenarios in which the monitor may be confronted with:

1. *Input interceptor sends an alert and the output interceptor detects an attack* - In this case the monitor has all the information to be certain that an attack has indeed taken place. The alert from the input interceptor (URI and warnings issued) is used to configure the vulnerability detector by extracting the parameters of the URI that caused the alert.

2. *Input interceptor does not send an alert and the output interceptor detects an attack* - The input interceptor issued a false negative but the output interceptor module was successful at detecting a malicious action. The monitor extracts the parameters from the URI and executes the vulnerability detector.

3. *Input interceptor sends an alert and the output interceptor does not detect an attack* - The input interceptor issued a false positive, as the web application is probably already protected from that attack or the malicious input was not enough to trigger a vulnerability. In this case the vulnerability detector may be called besides maintaining a log of undergoing activity.

4. *Input interceptor does not send an alert and the output interceptor does not detect an attack* - In this case there was no anomalous activity detected by either of these modules and as such no action is required.

In case of conflict, the decision of the output interceptor should overrule the input interceptor. This reasoning comes from the fact that the input interceptor is completely oblivious to the context of the code. However, since the output interceptor is positioned after the input processing, it has a more accurate representation of how the application reacts to the provided input.

Triggering Code Analysis

To adequately configure the vulnerability detector, the monitor extracts both the parameters and name of the file that is targeted by the request. The parameters serve to identify the entry points of the file and initiate the analysis. For example, from the request `http://example.com/login.php?user=user" or 1=1;-- &pass=foo` the parameters `user` and `pass` are extracted along with the name of the file `login.php`, which are used to limit the entry points that have to be considered and indicate the file to be analysed respectively.

3.2.5 Vulnerability Detector

The vulnerability detector module is responsible for performing a directed taint-based static analysis of the code that was targeted by the attack. As previously mentioned, the vulnerability detector is always configured by the monitor module prior to execution. As such, every execution considers only the parameters included in the packet that triggered the monitor to take action. In other words, there is never a generic execution of taint analysis, as each analysis is specific to the attack being attempted. Thus, contrarily to traditional static analysis, we aim to consider only the attack surface explored by the attack at hand. This way we execute an analysis focusing only on the values that would be tainted by the input provided in the attack. This results in a more focused effort to detect vulnerabilities, rather than a conventional static analysis in which every input value would be tainted and executed, allowing for more in-depth analysis within a given time interval. For instance, executing a ten millisecond analysis with a directed static analyser would yield better results than a conventional static analyser within the same amount of time. The vulnerability detector starts by interpreting the configuration provided by the monitor and then examines the code by following the parametrization.

Configuration

As previously mentioned, the vulnerability detector is configured by the monitor in order to follow the inputs. To do so, the detector sets up the entry point vectors of the program

with the received parameters, obtaining a list of the possible entry points, and the initial analysis file with the received filename. Considering the request `http://example.com/login.php?id=id" or 1=1;--` and the PHP superglobal variables, the filename `login.php` and the parameter `id` would be extracted, and then the `id` is used would be extracted and used to construct the following list with the possible entry points:

- $_SERVER['id']
- $_REQUEST['id']
- $_POST['id']
- $_GET['id']
- $_FILES['id']
- $_ENV['id']
- $_COOKIE['id']
- $_SESSION['id']

The list always includes all superglobal variables of PHP because: (1) it is not possible to know how the program retrieves the variables (i.e., `$_GET` and `$_REQUEST` fulfil similar purposes); (2) it is preferable to analyse the code slightly by excess, rather than overlook a possibly tainted superglobal variable with the same name.

In addition, the target file name is also provided by the monitor. Therefore, the detector will start the analysis over that file of the web application, looking for the above mentioned entry points.

Directed Static Taint Analysis

The analysis that we propose, instead of examining the entire program, it should be directed towards the code that involves variables that depend on the entry points of the application. In this sense, it is possible to make the analysis faster and more precise, since we are limiting the execution paths to be inspected by focusing only on the paths that start with the previously defined entry points and on an analysis that starts in a specific file. In addition, the vulnerability correction only has to consider the same execution paths to find out the location where the bug should be patched (see next section).

Figure 3.4 illustrates two possible execution paths of a program. Since only `Path 1` reaches a sensitive sink (e.g., *mysqli_query*), the goal is to taint only that path via the `$_GET['id']` entry point of the application. In this case, if a malicious request was sent with the `id` parameter defined, the vulnerability detector would only analyse `Path 1` and verify that it reaches a sensitive sink. On the other hand, if the `harmless` parameter is

received, the analysis would not detect an issue and would report in a log that the received warnings were likely false positives. The end results are two lists with details about the processed variables. One that contains the following information about the vulnerable variables: *variable name*, *vulnerability type*, *line*, *sensitive function* and *dependencies*. The second list exposes information about all parsed variables: *index*, *variable name*, *line*, *dependencies* and *tainted status*.

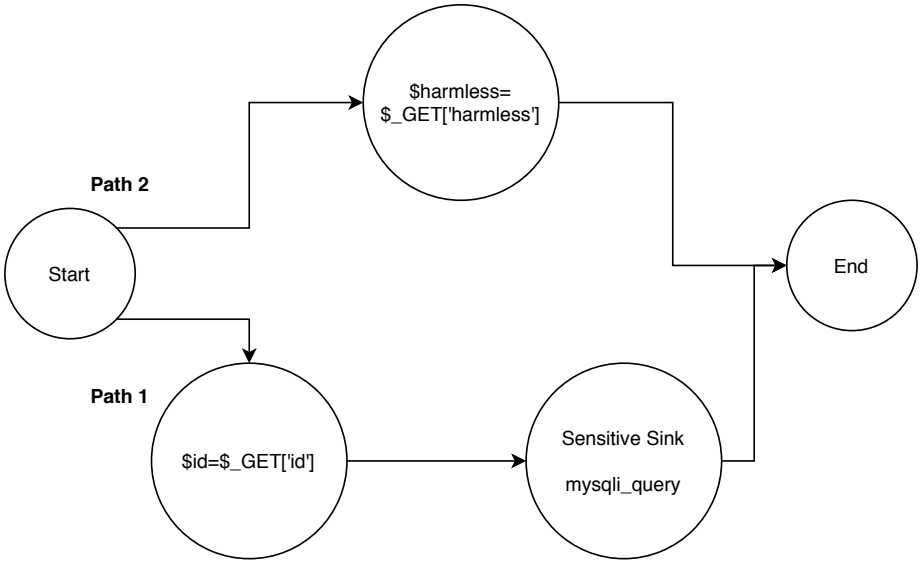

Figure 3.4: An example of path execution.

The meaning of these fields are as follows:

- *Index* - An index value that is incremented per each parsed variable, giving it a unique identifier;
- *Variable Name* - The name by which the variable is referenced within the code;
- *Line* - The line of code in which the variable can be found;
- *Dependencies* - The index of the variables which the current variable depends on;
- *Tainted status* - An indication on whether the variable has been tainted or not;
- *Vulnerability Type* - The type of vulnerability that the variable has incurred;
- *Sensitive Function* - The function that acts as sensitive sink and causes the vulnerability to happen.

3.2.6 Vulnerability Corrector

This module has the task of analysing the output from the vulnerability detector and patching the code by utilizing *code fragments* that employ sanitization functions. After the vulnerability detector is executed, it will output a list of all vulnerabilities that were found,

together with sufficient information to determine the respective taint paths. The vulnerability corrector iterates over these vulnerabilities and uses their details (vulnerability type, line of code and compromised variables) to identify which variables need to be sanitized and the lines of code that need to be altered.

While processing the discovered vulnerabilities, if any of these can be removed in the same line that the affected variable is found, it will immediately have their line and variable name marked for correction. However, some vulnerabilities cannot be corrected directly in the line where they are found. For instance, a SQL Injection has to be corrected before the query is formulated, otherwise it might make the query unusable. Therefore, in this case, the corrector first has to find the vulnerability in the list and check the type, and then it resorts to the dependencies to determine the tainted variables that are used to formulate the query. Next, it marks the line and variables that need to be sanitized. Finally, the module accesses the original file and copies the code line by line to a new file. When it reaches a line that was marked for correction, it will rewrite the line in order to apply a sanitization function over the tainted variable.

The execution of the vulnerability corrector results in a safer version of the application in question. A copy of the original file is always kept as a matter of precaution and for reference, allowing the programmers to understand how the vulnerability occurred.

3.3 Example of Execution Case

A representative example of an execution would be processed as follows:

1. A malicious request is sent from the user to the web application.

2. The request is captured by the input interceptor module, which is then checked for signs of malicious activity:

   (a) If the input interceptor detects any erroneous data, it generates an alert that is transmitted to the monitor;

   (b) If the input interceptor does not detect any malicious activity, only a copy of the request is sent to the monitor.

3. The output interceptor gets the message that is produced by the web application and investigates for signs of malicious activity:

   (a) If the traffic represents a known kind of attack (e.g., SQL Injection), the output interceptor contacts the monitor module alerting that a malicious action has indeed taken place. This confirms that there is an ongoing attack and that a vulnerability exists in the code and was exploited. For prevention, the query is dropped as a safeguard measure to stop the attack from taking place;

(b) If the traffic does not constitute an attack, the message is allowed to proceed.

4. Upon receiving the alert and the request, the monitor sees if they might indicate a vulnerability in the code:

   (a) If there is an alert issued from the input interceptor, and the output interceptor confirms that an attack is under way, the monitor configures the vulnerability detector to follow the taint flow of the input values from the request;

   (b) If there is no warning issued from the input interceptor but the output interceptor says that an attack is occurring, the monitor also sets up the vulnerability detector to follow the taint flow of the input values included in the initial request;

   (c) In any other case, there is no action taken besides maintaining a log of the activity.

5. The vulnerability detector is executed with the configuration set by the monitor. This means that a directed taint analysis is executed over the files that the input might spread across. Only the taint flows belonging to the input variables of the request will be considered. The result of this execution should be a list of any vulnerability found along with any data that might be required by the vulnerability corrector to remove these bugs from the source files (such as vulnerable execution paths and the variables included in them).

6. The vulnerability corrector iterates over the affected files and proceeds to copy them. If a line is known to include a vulnerability, it should be sanitized either locally or in a prior line. The result of this operation is a set of new source code files without the vulnerabilities detected previously.

Chapter 4

Implementation of the Tool

This chapter presents our current implementation of the proposed architecture, which supports the evaluation of the developed work. Section 4.1 describes the main components that were used in the proof of concept prototype, explaining in more detail the parts that had to be modified. Section 4.2 and 4.3 give an example of the use of the prototype to locate and correct a vulnerability.

4.1 Main Modules

The prototype is based upon the architecture that focuses on the input interceptor, and as such the output interceptor module was not implemented (Figure 3.2). Furthermore, the current solution is only programmed to detect and patch SQL Injection and XSS vulnerabilities. Figure 4.1 presents a diagram of the implemented architecture, while the remaining subsections detail how each of the represented components are built. The color coding for the diagram is as follows:

- Blue - Components that did not have to be altered. At most, these components had to be reconfigured to work properly with the other parts of the system;
- Red - Components that were implemented and modified to make vulnerability correction and detection possible;
- Green - The result of the execution of the architecture, which is a new safer version of the web application.

4.1.1 Input Interceptor

For the input interceptor a double proxy is used consisting of an Apache proxy server with ModSecurity web application firewall (WAF) v3.0 [41], using the Core Rule Set (CRS) 3, and a Java application that communicates with both the monitor and the web

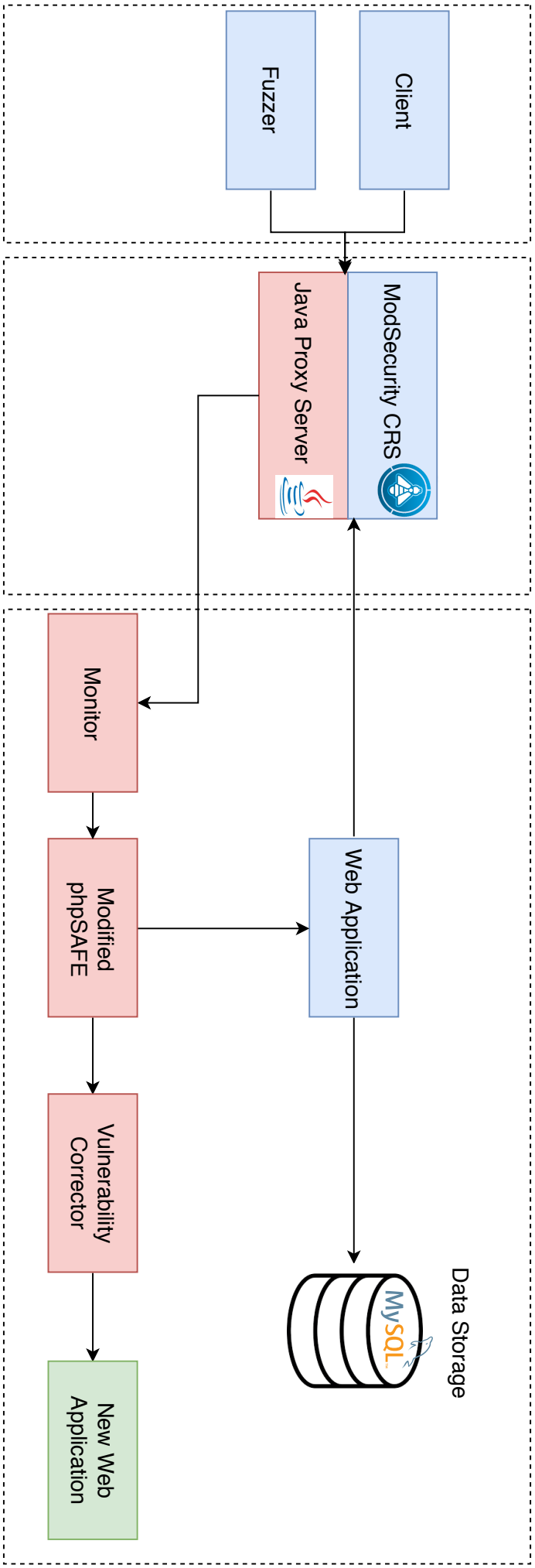

Figure 4.1: Current implementation of the proposed architecture.

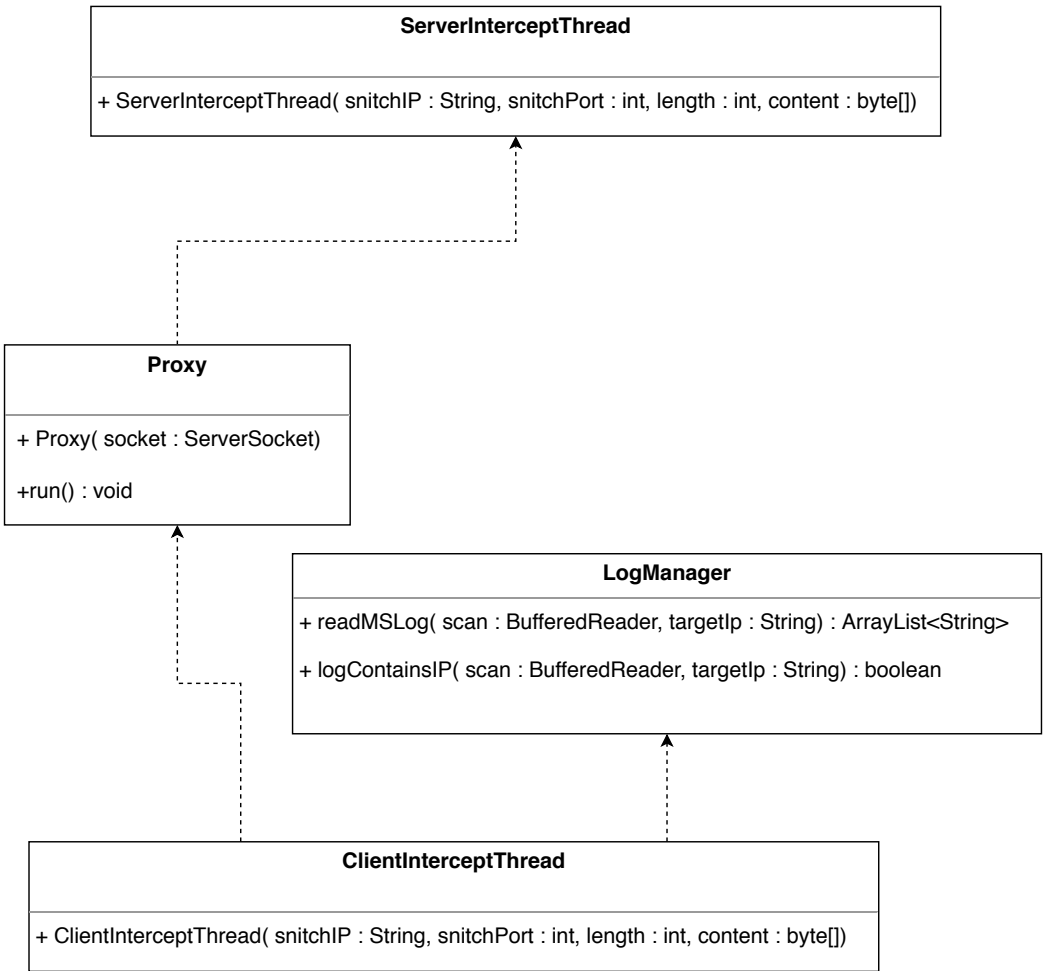

Figure 4.2: Input interceptor UML.

application. Warnings are generated by the Apache proxy with ModSecurity CRS 3 installed, where every request that is received is matched against a set of defined rules. If the request matches one or more of the installed criteria, one or more warnings are issued and stored in the error log of the Apache server. The default location of the log file is `/var/log/apache2/error.log`.

The Java application looks at the Apache error log for novel warnings by searching for entries that have been written by ModSecurity and whose activity matches the IP of the client that is under observation. Next, it combines both the packet and the warnings, to originate an alert that acts as an IoA. This IoA is then forwarded to the monitor and the packet is sent to the web application. The proxy can, however, easily be configured to block the request if that is desired. Note that these warnings can range from the detection of malicious input (i.e., SQLI and XSS) to displays of abnormal activity (such as using an IP directly as input).

Figure 4.2 shows a UML representation of the current implementation of the input interceptor. Each component is used as follows:

- *Proxy*
  - Proxy - The constructor of the class receives a ServerSocket as parameter, which is the socket that will be opened to allow for communication with clients.
  - run - The runnable method that executes the proxy server. It will establish a connection with the client and a connection with the web application server. It also starts two threads that are used to communicate with the monitor and provide information about the attacks (the two next components).
- *ClientInterceptThread*
  - ClientInterceptThread - Constructor of the thread. This thread establishes a connection with the monitor and sends a copy of the request that was received from the client, followed by the warnings that were generated by ModSecurity. This last information is obtained by accessing and parsing the data stored in `/var/log/apache2/error.log`.
- *ServerInterceptThread*
  - ServerInterceptThread - Constructor of the thread. This thread establishes a connection with the monitor and transmits a copy of the response that the server returns to the client.
- *LogManager*
  - readMSLog - This method inspects the log file where ModSecurity stores the warnings and checks for recent warnings related to a given IP.
  - logContainsIP - This method is a boolean check that determines if a certain IP is or is not contained within the log.

4.1.2 Monitor

The monitor is implemented as a Java application that establishes a connection with the Java application in the input interceptor. Each time that the monitor receives the warnings and the client request packet, it stores the collected information together with the request type, and in case of a POST, it also saves the user input. Next, the monitor examines the log and in case the warning includes the tag of a relevant vulnerability (for instance, SQL Injection), it edits a configuration file that belongs to the vulnerability detector. This gives an indication of the entry points that should be considered when analysing the program of the web application. Finally, it orders the execution of the vulnerability detector, specifying the name of the file in the HTTP request header as the target to be analysed. In case no name is provided, the detector automatically attempts to analyse the `index.php` file.

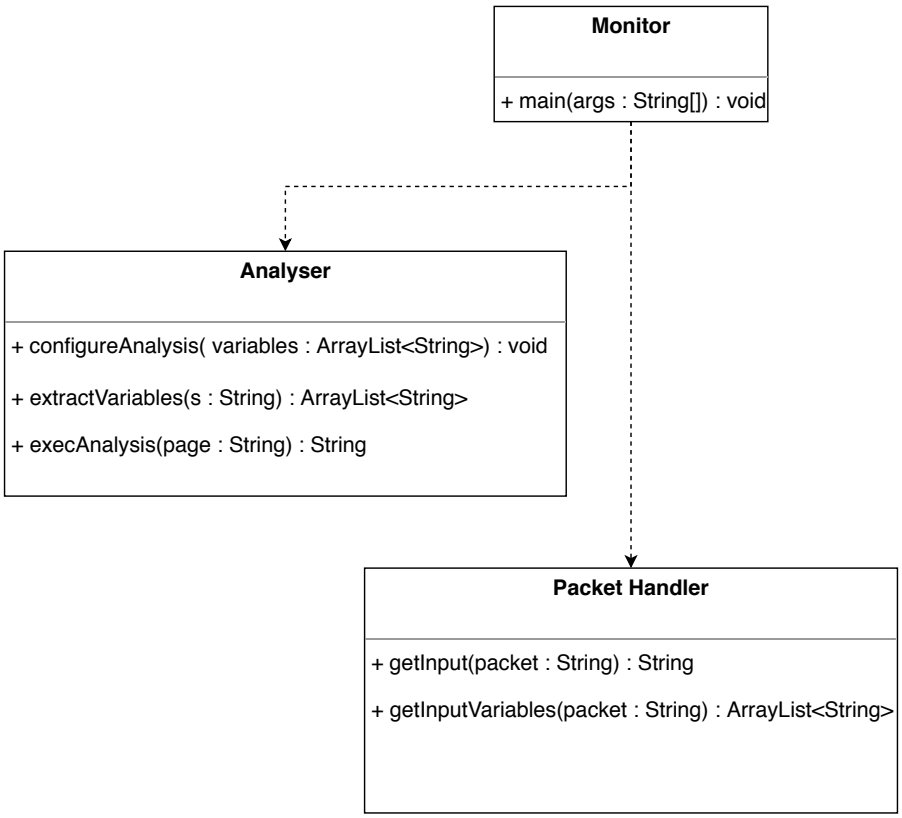

Figure 4.3: Monitor UML.

Figure 4.3 presents a UML of the classes that were developed for the implementation of the monitor. Each class and component have the following purpose:

- *Monitor*
  - main - The monitor class acts as a server. It waits for connection requests from the input interceptor. Once a communication from the input interceptor is received, the monitor makes a first assessment of the request type. This is done because the retrieval of parameters from POST type requests is treated differently from the GET type request. This last type allows the retrieval of data from URI. The URI and parameters with the respective values are written to a log file. Next, if there is also a warning sent by the input interceptor, the log entry is extended and the warnings are associated with the request. Finally, the monitor checks if any of the warnings is related to a known vulnerability (e.g., by searching for a tag relative to SQL injection). If this condition is verified, the parameter names are extracted and used to edit a configuration file for the vulnerability detector.
- *Analyser*
  - alterArray - This method changes the configuration array that is used by the

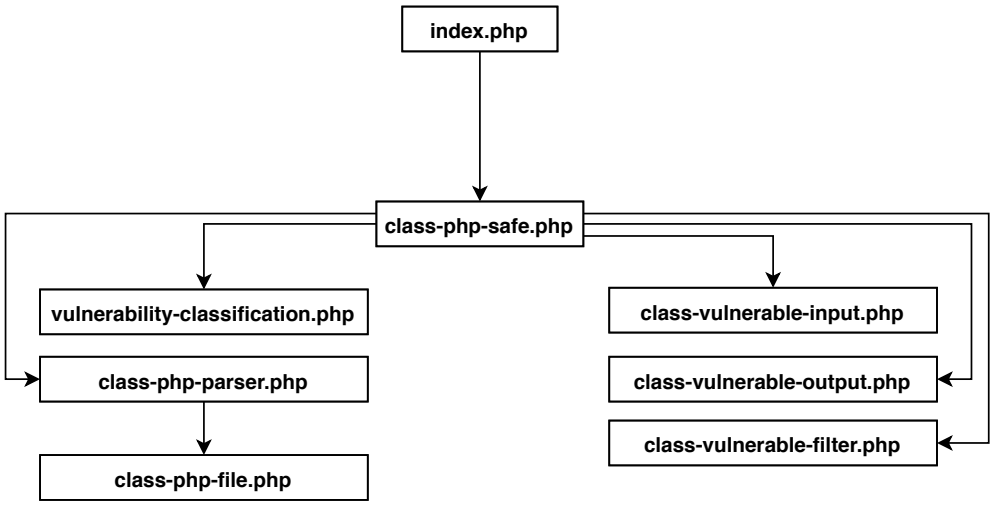

Figure 4.4: Simplified phpSAFE class graph.

vulnerability detector when setting up the tainted variables. By calling this method, the monitor is able to direct the analysis that is performed by the vulnerability detector.

- extractVariables - This method has the purpose of retrieving the variables (or parameters) included in the URI, which have been sent by the input interceptor.
- execAnalysis - This method retrieves the target file name and forges an HTTP request that will be sent to the vulnerability detector, saying which file needs to be analysed. It also initiates the execution of the vulnerability detector.

- *PacketHandler*
  - getInput - This method has the objective of obtaining only the input portion of a request (e.g., `<param>=<value>&...&<param>=<value>` format).
  - getInputVariables - This method receives the packet and extracts only the names of the parameters passed by the request, i.e., the entry point names (e.g., from `<param>=<value>&...&<param>=<value>`, the `<param>` are extracted).

4.1.3 Vulnerability Detector

The vulnerability detector is a PHP application based on phpSAFE [40], which statically analyses PHP code searching for XSS and SQLI vulnerabilities. Figure 4.4 shows a simplified graph with the classes of phpSAFE. Next, we provide a summary description of the represented classes and the modifications that were made in order to make directed static analysis possible:

- index.php - This is the index page of phpSAFE, which provides a visual representation of the results.
- class-php-safe - This is the main class of phpSAFE. It is responsible for calling the methods of class-php-parser in order to carry out the analysis of the code. This was the part of phpSAFE that was subject to changes:
  - *Vulnerable Input*: There were modifications to the function that indicated how the vulnerable input is parsed. Originally, phpSAFE would only consider a superglobal type (e.g., `$_GET`). However, the goal of the directed static analysis is to only consider specific entry points of the program via the parameters included in the HTTP request. To this end, the parsing of vulnerable input was altered in order to find exact matches of both the superglobal type and the parameters that are passed (e.g., `$_GET['parameter']`).
  - *Correction*: Since the final goal of the tool is to correct the vulnerabilities, a new function was added to class-php-safe with the goal of: (1) identifying which variables within the the execution path should be corrected; (2) creating a copy of the original file; (3) the inserting *code fragments* in the application program (see next section). The code fragments are small PHP functions we developed that use sanitization functions to correct user inputs.
- class-php-file - This class is responsible for converting the code into tokens, which are used to generate an AST that supports static analysis.
- class-php-parser - This class parses the contents of the AST generated by class-php-file in order to understand the flow of the tainted input.
- class-vulnerable-input.php - This class contains a list of the input that should be considered vulnerable. This file is rewritten by the monitor to consider only the input that generated alerts, thus directing the static analysis.
- class-vulnerable-output.php - This class contains a list of sensitive output sinks that are known to cause bugs.
- class-vulnerable-filter.php - This class contains a list of sanitizing functions to keep track if any input is sanitized in the code, allowing to change the attribute associated with the variables from tainted to untainted.
- vulnerability-classification.php - This file contains the definition of various constants that are used during analysis, such as macros for vulnerability classification.

In summary, the analyser works as follows: First, the vulnerability detector is configured by using the data provided by the monitor module, which dictates the variables of

the program that should be set as tainted. Next, a model of the program is constructed by using the PHP `token_get_all` function, which splits the code into tokens. These tokens are used to generate an AST by following the paths in the code. Each of the tokens is either an array that contains the token identifier, the value of the token and line number of the script, or a string that represents the code semantics. As per the analysis of the code itself the flow of each variable and their respective data required for taint analysis is stored in a multi dimensional array. This array is generated by following the flow of tainted variables. Once the execution is terminated, three lists are returned:

- *Vulnerable Variables* - A group of variables that were found to be vulnerable.
- *Parser Variables* - A list of all the variables that were parsed during the analysis.
- *Output Variables* - A set of variables that reach an output sink of the application.

These lists share similar information about each of the variables:

- index - Index of the variable in the current list;
- name - The name of the variable;
- scope - The scope of the variable (either global or local);
- input - Set if the variable is an input;
- output - Set if the variable is an output;
- file - The file name in which the variable is located;
- line - The line in which the variable is located;
- vulnerability_type - The type of the detected vulnerability (only for Vulnerable Variables);
- dependencies_index - The indexes of the dependencies of a variable by order of appearance in the parser variables list;
- start_index - The index of the token in which the variable started being declared;
- end_index - The index of the token in which the variable declaration was finalized.

It is important to collect more than just the vulnerable variables for the corrector, as we want to apply the sanitisation function before the execution reaches the sensitive sink (e.g., a SQL query is formed). For this, we need to be able to follow the taint path until the last line that has influence over the variable before it is used in the sensitive sink. Two crucial lists are built to get this knowledge, the vulnerable variables and parser variables,

where the former is required to identify the occurrence of a vulnerability and the latter is able to determine which variables cause the vulnerability that needs to be fixed.

Some alterations had to be made to achieve this goal. Although phpSAFE includes a file that allows for the configuration of vulnerable input, it was only designed to consider superglobal variables. For example, it was possible to set all `$_GET` variables as tainted but it unsupported the ability to explicitly declare a `$_GET['param']` variable as tainted. The code was changed to allow for this type of definition within the configuration file provided by the monitor.

Finally, a log file that registers the vulnerabilities that were found is generated, detailing the bugs that were identified. This is done to leave a record of the analysis and to mitigate any false positive that might have been issued by the input interceptor. If the input interceptor and the monitor required a certain analysis but it did not result in the discovery of vulnerabilities, a confirmation of their non-existence might still be required.

4.1.4 Vulnerability Corrector

The vulnerability corrector is a piece of PHP code within the vulnerability detector that accesses the results from static analysis. To achieve this, a new method was added to phpSAFE. This method should be executed when the analysis is finalized (i.e., before displaying results) to ensure that the corrector module has all the information required to apply the fix to the code. Additionally a new file was added to phpSAFE that acts as a correction library. It includes a variety of functions that can be employed to sanitize input.

The corrector iterates over the vulnerable variables list and identifies the bugs by type. If the type is SQL Injection, it utilizes the dependencies to reach the tainted variable that is included in the SQL query. This has to be done in order to ensure that the variables that influence the query are sanitized and not the query itself, as modifying the query could make it unusable and cause an error once the DBMS tries to process it. This is done for every vulnerability found, where every variable that requires sanitization is added to a list that contains: (1) the name of the variable; (2) the line in which the variable needs to be sanitized.

Once the corrector is done processing the variables and constructing the list, it will create a new file with a copy of the code from the original vulnerable file. While copying the contents, the corrector also includes the library with the code fragments and checks the line number for any variable that might need fixing. Once such a variable is encountered, the corrector replaces the variable with a call to the sanitizing function (e.g., `$vulnerable` becomes `san_sqli($vulnerable)`).

Finally, the corrector will swap the original vulnerable file with the new safer version. The original code is never discarded, but instead it is renamed to `<original_file_name.php>_old`. This is done as a precaution in case of an error with the correction, and to let the programmers and administrators keep track of how the vulnerabilities occur and

to avoid future mistakes.

4.2 Usage Example

An execution of our current prototype follows the subsequent order:

1. A client (or fuzzer) sends a potentially malicious HTTP request to the web application.

2. The input interceptor module captures the request before it reaches the web application and the ModSecurity CRS Apache proxy matches the request with the built-in rules. If necessary, it generates the warnings, storing them in the system log.

3. Still within the input interceptor module, the Java application accesses the log to retrieve the warnings and combine them, to generate a custom alert that is transmitted to the monitor.

4. Upon the arrival of the alert, the monitor analyses the log for any tag that indicates a possible vulnerability within the code (namely SQL Injection). The monitor then edits the configuration file for the vulnerability corrector to consider only as tainted the input fields of that request.

5. The vulnerability detector statically analyses the code addressing only the paths tainted by the fields specified by the monitor in the configuration file. Once the execution is finished, the detector outputs two lists fundamental for correction: one including all the vulnerable variables and one containing all the parsed variables during the process. All parsed variables that were tainted are tagged as such and have their dependencies scrutinized.

6. If any bugs are found during the analysis, the vulnerability corrector accesses the files that have the errors and starts to copy them line by line. Once the last tainted line before the vulnerability is reached, the affected variable is sanitized, thus removing the impact any malicious input.

The final result of this execution is a new version of the web application without the vulnerabilities detected. All this process is performed at runtime, meaning that the web application continues to be online throughout the whole analysis and patching.

Two key results of the current prototype are that it demonstrates the possibility of directed taint analysis by providing specific inputs and that it is achievable to detect and patch vulnerabilities inside a web application at runtime.

The conjugation of all these components enables six different outcomes, depending on the precision of the input interceptor and the vulnerability detector. In this sense we can contemplate the following scenarios of execution:

- *The input interceptor generates a false positive:*
  - The vulnerability detector wrongly identifies a vulnerability: Both components produced a false positive. Although this case is not optimal, there are no adverse consequences because the inserted corrections are harmless and do not jeopardize the application operation.
  - The vulnerability detector does not find flaws in the code: This is a positive result, as it avoids loss of time by the programmer/system administrators to search for an issue that does not exist.
- *The input interceptor generates a true positive:*
  - The vulnerability detector finds a vulnerability: Both components are in agreement, meaning that probably the alert generated by the input interceptor is relevant in the context of the application and it revealed a vulnerability in the target file that was corrected.
  - The vulnerability corrector does not discover any vulnerability: This is the worst case scenario. The input interceptor generates a correct alert, but the detector does not find any problem in the target file, causing it to remain vulnerable. To minimize this situation, is if preferable to do a more careful analysis, even if it has to be done at a later moment to avoid the unavailability of the service due to time constraints.
- *The input interceptor generates a true negative:* The traffic is considered normal, there is no attack or vulnerability associated with the input.
- *The input interceptor generates a false negative:* The input interceptor erroneously labels an attack as normal input. This does not activate the code analysis and allows for an existing vulnerability to be exploited. To mitigate this problem it would be required that the input interceptor has a better judgement over the input, but such is outside of the scope of this dissertation.

4.3 Detection and Correction Example

To exemplify the detection and correction of vulnerabilities, let us consider the following piece of code from the `test.php` file, which contains both a XSS and a SQLI flaw.

```
1 $id=$_GET['id'];
2 $query="SELECT first_name, last_name FROM users WHERE user_id = '$id';";
3 $result=mysqli_query($query);
4 .
5 .
6 .
7 $echo $_GET['cookie'];
```

Also consider that an attacker has sent the following malicious request that attempts to exploit both vulnerabilities:

```
www.example.com/test.php?id=a' UNION SELECT "text1","text2";-- &Submit=Submit&cookie=
        <script>alert("hacked")</script>
```

This request is intercepted by the input interceptor and analysed with a set of rules, which detect the attack and generate an event according to the rules that matched. Next, the input interceptor creates an alert based on the events and the request that it captured, including one or more `[tag]` that explain the observed incidents. The alert is then sent to the monitor.

```
[tag "attack-sqli"] [uri "www.example.com/test.php?  id=a'
 UNION SELECT "text1","text2";-- -&Submit=Submit
 &cookie=<script>alert(hacked)</script>"]

[tag "attack-xss"] [uri "www.example.com/test.php?
id=a' UNION SELECT "text1","text2";-- -&Submit=Submit
&cookie=<script>alert(hacked)</script>"]
```

Once the monitor receives the alert, it consults the `[tag]` to interpret the alert and the nature of the possible vulnerability. Next, it extracts the `id` and `cookie` parameters from the URI, which correspond to the entry points of the application, and the name of the file `test.php`. This data is used to configure the execution of the vulnerability detector so that it only considers the specified entry points during the analysis. By examining the `test.php` file, both `$id` and `$_GET['cookie']` become tainted by the `$_GET` vector. The `$id` variable is later used to formulate a SQL query, which is then executed by the function (and sensitive sink) `mysqli_query`. This results in the detection of a SQLI vulnerability. The last line also incurs in a vulnerability, since the `echo` function (and also a sensitive sink) is called indiscriminately over the superglobal variable `$_GET['cookie']`. Therefore, a script can be included in the cookie, thus resulting in a XSS vulnerability. Finally, the detector creates a list of vulnerable variables, which associate to each bug the following information:

```
Index: 5
Vulnerability: SQLI
Vulnerable Variable: $query
Dependency: 2
Function: mysqli_query
Line: 3

Index: 6
Vulnerability: XSS
Vulnerable Variable: $_GET['cookie']
Dependency: ——
Function: echo
Line: 7
```

In addition, it returns a list of the parsed variables with the following information:

```
Index: 0
Variable: $id
Status: Tainted
Line: 1
Dependency: 1
```

```
Index: 1
Variable: $_GET['id']
Status: Tainted
Line: 1
Dependency: ——

Index: 2
Variable: $query
Status: Tainted
Line: 1
Dependency: 3

Index: 3
Variable: $id
Status: Tainted
Line: 2
Dependency: 0

Index: 4
Variable: $result
Status: Untainted
Line: 3
Dependency: ——

Index: 5
Variable: $query
Status: Tainted
Line: 3
Dependency: 2

Index: 6
Variable: $_GET['cookie']
Status: Tainted
Line: 7
Dependency: ——
```

The vulnerability detector then activates the vulnerability corrector by providing both lists. The corrector identifies the type of the flaw. As there is a SQL Injection vulnerability present, the corrector checks the dependencies to determine where the sanitization function should be called. In this case, it tracks the dependency of `$query` at index 5 to `$query` at index 2, which will then lead to `$id` at index 3. This last one is the variable that is the target of the sanitization. As for the XSS vulnerability, since it does not have any dependencies, it is safe to apply the fix directly over the vulnerable variable.

Let us consider that `san_sqli` is a sanitization function that has the capability of removing SQLI vulnerabilities. In this case, this function would be used to correct the `$id` variable. This results in the following version of the program that removes the SQLI vulnerability.

```
1 $id=$_GET['id'];
2 $query="SELECT first_name, last_name FROM users WHERE user_id = '.san_sqli($id).';";
3 $result=mysqli_query($query);
4 .
5 .
6 .
7 $echo san_xss($_GET['cookie']);
```

Chapter 5

Evaluation

The main objective of the experimental evaluation is to test and verify the efficiency of the platform at detecting and correcting SQLI and XSS vulnerabilities of PHP applications. In this sense the following questions were asked:

1. *Is the platform capable of detecting vulnerabilities of both classes?*
2. *Is the platform capable of correcting vulnerabilities?*
3. *Does the vulnerability detector have the capability to identify false positives from the input interceptor?*

Testing was done across 5 web applications, and a total of 145 files were analysed. To avoid having to rely on a ”perfect fuzzer”, which is beyond the scope of this thesis, the testing was accomplished by first doing a manual audit of the code and determining the input fields of each application. Each input was setup in order to specifically trigger the event generation by ModSecurity, which issues a warning at any given malicious input. This can result in false positives as ModSecurity does not have any context of the application. However, the results of the subsequent static analysis are able to mitigate the consequences of a false positive by providing confirmation or denial of the existence of any issue within the code. The following sections are dedicated to present the testbed and explain the tested web applications, with the two final sections giving a general overview of the results and illustrating examples of code correction.

5.1 Testbed

The current prototype was validated in a testbed composed of three virtual machines (VM), as displayed in Figure 4.1. The components were distributed in the following manner:

- Client/Fuzzer (VM1) - Kali Linux with the default set of tools. This allows the component to act as a regular client while also including fuzzing tools.

- Input Interceptor (VM2) - Xubuntu 16.04 with an Apache proxy server using ModSecurity CRS 3 for warning generation and a Java proxy server that communicates with the monitor and the web application.
- Web Application and other components (VM3) - Xubuntu 16.04 with an Apache server to run the web application, vulnerability detector and vulnerability corrector as well as the monitor via a Java application.

5.2 Web Applications

This section provides some context about the tested web applications along with a brief analysis of the obtained results. Table 5.1 gives a summary of the results of the experiments.

5.2.1 DVWA

The first web application used for testing was DVWA (Damn Vulnerable Web Application) [1], an application that intentionally includes various vulnerabilities of several kinds and across different criticality levels. The main purpose of this application is to provide a platform for security professionals to test their skills, help developers understand the processes required to secure an application, and expose vulnerability cases to students for academical purposes.

DVWA was chosen as a starting point because it is a simple application that allows for an easy and fast analysis of the code. Although there are many vulnerability examples, we focused only on the SQLI because it allowed for testing of both detection and correction. The implementation of our monitoring tool is able to successfully detect and correctly patch the SQL Injection vulnerability that is present within the code.

5.2.2 ZeroCMS

The second web application used for testing was ZeroCMS [5], an application that provides a content management system. More specifically, it has the goal of sharing articles between various users registered within the platform.

Web application	Total File	LoC	Attacks	File Anal.	LoC Anal.	Total Vuln.	SQLI	XSS	FP	FN	File. Corr.
DVWA	343	19.142	1	1	24	1	1	0	0	0	1
ZeroCMS	29	1.001	14	14	736	18	13	5	0	0	6
AddressBook	238	28.526	63	63	8280	29	7	22	0	9	4
WebChess	28	3.634	28	28	3634	122	75	47	0	1	12
refbase	156	52.200	39	39	25549	4	0	4	0	0	0
Total	**797**	**104.503**	**145**	**145**	**38.223**	**174**	**96**	**78**	**0**	**10**	**23**

Table 5.1: Informational table for each web application.

This application is composed of a total of 29 files from which 14 PHP code files were tested. Out of these, 6 were found to be vulnerable. All 6 included a SQLI vulnerability and 3 of them were also vulnerable to XSS. The tool was able to correct all the files and mitigated the false positives issued by the input interceptor.

5.2.3 AddressBook

The third web application was AddressBook [2], an application that allows users to maintain an agenda of their contacts and organize them by groups. This application includes a total of 238 files out of which 63 were analysed. During this test, phpSAFE failed to generate the AST and parse the code. We suspect this might have occurred because the `token_get_all` function was unable to split the code into tokens appropriately.

The results from these tests revealed 9 false negatives as a consequence from the AST not being generated properly. This means that the input interceptor issued an alert that was correct, a vulnerability was present within the code, but it was not detected or corrected. The remaining results eliminated the false positives and detected 7 SQLI and 22 XSS vulnerabilities in 16 files (from a total of 25 vulnerable files).

5.2.4 WebChess

The fourth test was done with WebChess [4], a web application that implements a chess game that can be played between two users. Out of all the tested applications, WebChess was the most vulnerable with 122 vulnerabilities, out of which 75 were SQLI and 47 were XSS.

Like in the previous tested application, there was a single occurrence of the analysis that was not successful due to the AST not being generated properly. This did not cause a false positive as the file that was the victim of this failure was not vulnerable. However, there was a case in which the static analysis did not perform as expected, which caused 1 false positive. The remaining files were all analysed correctly with 13 of them being vulnerable. In total, 28 files were subject to testing.

5.2.5 refbase

The final application to be tested was refbase [3], a service that allows users to manage bibliographic references in various formats. This application proved to be the least vulnerable as it included a visible effort to sanitize SQL queries. Therefore, no SQLI was found by the manual analysis that was made.

The application spans over 156 files out of which 39 were tested. The results revealed that 2 files suffer from a total of 4 XSS vulnerabilities. Once again, the static analysis proved successful at mitigating the effects of false positives issued by the input interceptor.

	Input Interceptor	Vulnerability Detector		Vulnerability Corrector
True Positive	49	Detection	39	23
		Error	10	0
False Positive	96	Detection	0	0
		Error	96	0

Table 5.2: Table of analysed files.

Additionally 2 of the 4 discovered XSS vulnerabilities were zero-day (vulnerabilities that had not been previously reported).

5.3 Experimental Results

Overall, 145 files were tested. Since every client request issued was an attack designed to explore a few entry points, ModSecurity generated a warning for every attempt. This resulted in 49 true positives found by the input interceptor. The static analysis was able to confirm that 39 vulnerable files existed containing 174 bugs. In the remaining 10 cases, the static analysis was unable to generate the AST because the PHP function `token_get_all` failed to extract the tokens from the code.

From the 39 confirmed vulnerable files 23 were patched. This corresponds to the totality of files that were affected by SQL Injection vulnerabilities. As per the remaining 96 false positives, the static analysis was able to effectively confirm the non-existence of any vulnerable file in every case. Thus, this mitigates the consequences of the lack of code context on the input interceptor.

The final results of testing resulted in a positive response of the three questions that were raised: (1) *Testing shows that the current implementation accurately detects vulnerabilities (see Table 5.1).* (2) *Testing proves that the tool is capable of correcting vulnerabilities (see Table 5.1 and Table 5.2).* (3) *The Vulnerability Detector is capable of reducing the number of false positives issued by the Input Interceptor (see Table 5.2).*

5.4 Correction Examples

This section provides a few examples of the corrections that were made in some web applications.

5.4.1 ZeroCMS

This case is rather peculiar. There are several entry points that could potentially be used to do an SQL Injection. The developers sanitized most of them correctly but seemingly forgot to sanitize one of the variables.

In the following example `$user_id` is tainted directly by the `$_POST` vector. The remaining variables are also tainted but they go through validation procedures (the `mysql_real_escape_string` function). The applied fix performs sanitization validation over the `$user_id` variable, removing the characters that typically are utilized to perform an SQLI attack (line 16).

```
1  $user_id = (isset($_POST['user_id'])) ? $_POST['user_id'] : '';
2  $email = (isset($_POST['email'])) ? $_POST['email'] : '';
3  $name = (isset($_POST['name'])) ? $_POST['name'] : '';
4  $access_level = (isset($_POST['access_level'])) ? $_POST['access_level']
5                : '';
6
7  if (!empty($user_id) && !empty($name) && !empty($email) &&
8              !empty($access_level) && filter_var($email, FILTER_VALIDATE_EMAIL)
9              && preg_match('/^[A-Za-z0-9]{1,255}$/', $name))
10                          {
11 $sql = 'UPDATE `zero_users` SET
12                       email = "' . mysql_real_escape_string($email, $dbx) . '",
13                       name = "' . mysql_real_escape_string($name, $dbx) . '",
14                       access_level = "' . mysql_real_escape_string($access_level, $dbx) . '"
15                   WHERE
16                       user_id = ' . san_sqli(1,$user_id);
17
18 mysql_query($sql, $dbx) or die(mysql_error($dbx));
```

5.4.2 AddressBook

In this case the developers seemingly attempted to make the code safer by utilizing the `strip_tags` function. However, this function only removes HTML, XML and PHP specific tags. This means that an SQL Injection is still possible as it does not depend on the usage of these tags to be executed. The password (`$urlpw`) that is passed by the client is encoded in MD5. Although it is weak hash algorithm, since the password is encoded, it is not usable to inject code into the database. However, the `$urlun` variable is not subject to any other sanitization aside from the `strip_tags` function. This causes the variable to reach the sensitive sink unsanitized. The applied fix escapes any character that may be used to perform an SQL Injection (line 12).

```
1     mysqli_connect( $db_host, $db_username, $db_password )
2        or die( "Error! Could not connect to database: " . mysqli_error() );
3
4     mysqli_select_db( $db )
5        or die( "Error! Could not select the database: " . mysqli_error() );
6
7  $urlun = strip_tags(substr($_REQUEST['username'],0,32));
8  $urlpw = strip_tags(substr($_REQUEST['password'],0,32));
9
10 $cleanpw = md5($urlpw);
11
12 $sql="SELECT * FROM users WHERE username='san_sqli(1,$urlun)' and password='$cleanpw'";
13
14  $result=mysqli_query($db,$sql);
```

5.4.3 WebChess

This particular case is interesting for two reasons: (1) the cause of the vulnerability is not a variable but instead it is a session parameter that was tainted; (2) the session parameter is tainted in another file and propagates to the entire session.

WebChess is an extremely vulnerable application. This is partly due to the lack of care that the developers had while formulating queries. There are multiple instances in which queries are defined directly with PHP superglobal variables without any protection.

The following is a simple example that was repeated often throughout the code of WebChess. The `$_SESSION` superglobal is used in query formulation. However, as it can be seen in the second piece of code (which is included in a different file), it depends on a `$_POST` superglobal variable. This means it can be easily exploited by an attacker. In the first file the `$_SESSION['gameID']` is tainted directly by user input via the `$_POST` vector. The session value is then used directly in a query. The applied correction sanitizes `$_SESSION['gameID']` as it is used to formulate the query, thus preventing the vulnerability exploitation (line 12).

```
1         if (isset($_POST['gameID']))
2                 $_SESSION['gameID'] = $_POST['gameID'];
```

```
1  function createNewGame($gameID)
2  {
3          global $CFG_TABLE;
4          global $numMoves;
5
6          if (!minimum_version("4.1.0"))
7                  global $_POST, $_GET, $_SESSION;
8
9          $numMoves = -1;
10         mysql_query("DELETE FROM " . $CFG_TABLE[history] . "
11         WHERE gameID = ".san_sqli(1,$_SESSION['gameID']));
12
13         initBoard();
14 }
```

Chapter 6

Conclusion

This thesis presents an architecture for automatic vulnerability detection and correction of PHP web applications at runtime by utilizing *directed static analysis* and *code fragments*. The architecture makes use of an input interceptor module that captures the request sent from a user to the web application and scans it for signs of malicious activity. An output interceptor that analyses the output of the web, a monitor that receives the data from both input and output interceptor. Next, the monitor orders the execution of a vulnerability detector module that will scan the code for vulnerabilities based on the injected inputs that will activate the entry points of the application. Finally, a vulnerability corrector that will correct the vulnerabilities detected.

The current implementation uses input interception to detect malicious input and direct the static analysis to discover vulnerabilities and correct them. The prototype was tested with 5 web applications where it was able to detect over 170 vulnerabilities. It proved effective at removing flaws and discovered 2 zero-day vulnerabilities.

6.1 Future Work

For future work, we believe that these results are indicative that a full implementation of the proposed architecture would make the analysis more effective as it becomes possible to analyse the parts of the program only when it is proven that they might be vulnerable (i.e., a malicious query flows through the web application and is captured before the reaches the data storage). Not only does this avoid unnecessary execution of static analysis but it also allows to detect eventual false positives that are issued by the input interceptor. Another aspect that can still be improved over the current prototype is the implementation of the detection and correction of other types of vulnerabilities. These two mentioned improvements would greatly increase the coverage of the tool and efficiency.

Bibliography

[1] Damn vulnerable web application. www.dvwa.co.uk/.

[2] Php addressbook. https://sourceforge.net/projects/php-addressbook/.

[3] refbase. http://www.refbase.net/index.php/Web_Reference_Database.

[4] Webchess. https://sourceforge.net/projects/webchess/.

[5] Zerocms. https://sourceforge.net/projects/zerocms/.

[6] Indicators of attack (IoA), 2014. https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-indicators-of-attack.pdf.

[7] jsfunfuzz, 2018. https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz.

[8] A. Alhuzali, B. Eshete, R. Gjomemo, and V.N. Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652, 2016.

[9] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves. Vulnerability discovery with attack injection. *IEEE Transactions on Software Engineering*, pages 357–370, May 2010.

[10] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2010.

[11] M. Böhme, V.T. Pham, M.D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

[13] C. Cadar, P. Godefroid, S. Khurshid, C.S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the International Conference on Software Engineering*, pages 1066–1071, 2011.

[14] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, pages 82–90, February 2013.

[15] D.U. Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Infomation sharind and Analysis Center (E-ISAC)*, 2016.

[16] B. Caswell, J. Beale, and A.R. Baker. *Snort Intrusion Detection and Prevention Toolkit*. 2007.

[17] O. Catakoglur, M. Balduzzi, and D. Balzarotti. Automatic extraction of indicators of compromise for web applications. In *Proceedings of International Conference on World Wide Web*, pages 333–343, 2016.

[18] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand. SOFIA: An automated security oracle for black-box testing of SQL-injection vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 167–177, Sept 2016.

[19] Q.A Chen, Z. Qian, Y.J Jia, Y Shao, and Z.M. Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 388–400, 2015.

[20] A.S Christensen, A. Møller, and M.I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the International Conference on Static Analysis*, pages 1–18, 2003.

[21] J. Dahse and T. Holz. Simulation of built-in PHP features for precise static code analysis. In *Proceedings of the Symposium on Network and Distributed System Security*, 2014.

[22] A. Dalai and S. Jena. Neutralizing SQL injection attack using server side code modification in web applications. *Security and Communication Networks*, pages 1–12, February 2017.

[23] F. Duchene, S. Rawat, J. Richier, and R. Groz. Kameleonfuzz: Evolutionary fuzzing for black-box XSS detection. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, pages 37–48, 2014.

[24] F. Duchène, S. Rawat, J. L. Richier, and R. Groz. LigRE: Reverse-engineering of control and data flow models for black-box XSS detection. In *Proceedings of Working Conference on Reverse Engineering*, pages 252–261, Oct 2013.

[25] T.K. George and P. Jacob. A proposed architecture for query anomaly detection and prevention against SQL injection attacks. *International Journal of Computer Applications*, pages 11–14, March 2016.

[26] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the USENIX Conference on Security*, pages 49–64, 2013.

[27] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, pages 445–458, 2012.

[28] Ibéria Medeiros. OWASP WAP - web application protection project, January 2018. https://www.owasp.org/index.php/OWASP_WAP-Web_Application_Protection.

[29] Imperva. The state of web application vulnerabilities in 2017. https://www.imperva.com/blog/2017/12/the-state-of-web-application-vulnerabilities-in-2017/, December 2017.

[30] T. Jensen, H. Pedersen, M.C. Olesen, and R.R. Hansen. THAPS: Automated vulnerability scanning of php applications. In *Proceedings of the Nordic Conference on Secure IT Systems*, pages 31–46, 2012.

[31] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, pages 27–36, 2006.

[32] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah. Acing the IOC game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 755–766, 2016.

[33] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou. SQLProb: A proxy-based architecture towards preventing SQL injection attacks. In *Proceedings of the ACM Symposium on Applied Computing*, pages 2054–2061, 2009.

[34] H.Y Lock and A. Kliarsky. Using IOC (indicators of compromise) in malware forensics. *SANs Institute*, 2013.

[35] P.D Marinescu and C. Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the International Conference on Software Engineering*, pages 716–726, 2012.

[36] I. Medeiros, M. Beatriz, N. Neves, and M. Correia. Hacking the DBMS to prevent injection attacks. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, pages 295–306, 2016.

[37] I. Medeiros, N. Neves, and M. Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, pages 54–69, March 2016.

[38] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch. FlowWatcher: Defending against data disclosure vulnerabilities in web applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 603–615, 2015.

[39] Naked Security by SOPHOS. Hackers sentenced for sql injections that cost $300 million, June 2018. https://nakedsecurity.sophos.com/2018/02/19/hackers-sentenced-for-sql-injections-that-cost-300-million/.

[40] P. J. C. Nunes, J. Fonseca, and M. Vieira. phpSAFE: A security analysis tool for OOP web application plugins. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 299–306, June 2015.

[41] OWASP. OWASP modsecurity core rule set (CRS). https://modsecurity.org/crs/.

[42] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 653–665, 2016.

[43] S. Rauti, J. Teuhola, and V. Leppänen. Diversifying SQL to prevent injection attacks. In *Proceedings of the IEEE Trustcom/BigDataSE/ISPA*, pages 344–351, Aug 2015.

[44] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, pages 1–16, 2016.

[45] VirusTotal. YARA the pattern matching swiss knife for malware researchers. http://virustotal.github.io/yara/.

[46] W3Tech. Usage statistics and market share of PHP version 5 for websites. https://w3techs.com/technologies/details/pl-php/5/all.

[47] W3Techs. World wide web technology surveys, June 2018. https://w3techs.com.

[48] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the International Conference on Software Engineering*, pages 171–180, 2008.

[49] J. Williams and D. Wichers. OWASP Top 10 2017 – the ten most critical web application security risks, 2017.