UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Ciências
ULisboa

ATTACKING WEB APPLICATIONS FOR DYNAMIC DISCOVERING OF VULNERABILITIES

João Manuel da Silva Caseirito

Mestrado em Segurança Informática

Dissertação orientada por:

Professora Doutora Ibéria Vitória de Sousa Medeiros

2021

Acknowledgements

Firstly, I would like to thank my advisor, Prof. Ibéria Medeiros for all the time invested in me and this dissertation, and all the motivational speeches that allowed me to keep going and finally conclude this project.

To my girlfriend Rafaela that has accompanied me for longer than these five years and to whom I am very grateful.

To you Avelãs, with whom I shared almost all my experiences in this long journey and to the remaining PSI14 members, Paulo, Ginja, Yoda and Audi, and also to Primeiro and Adaga, for the amazing time with all you guys, it has been a blast.

Finally, to my mother, Aida, and aunt, Teresa, for all the unconditional support over the years.

This work was partially supported by the national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference to PTDC/CCI-INF/29058/2017, project SEAL, and LASIGE Research Unit (UIDB/00408/2020), and through P2020 with reference to LISBOA010247FEDER039238, XIVT project, an ITEA3 European project (I3C4-17039).

Resumo Alargado

Atualmente, a Internet faz parte do nosso dia a dia, sendo que as aplicações são criadas sobre a mesma. Estas aplicações oferecem múltiplos serviços que são usados para resolver diversos problemas na nossa vida, tal como falar com amigos nas redes sociais, comprar bilhetes para a nossa próxima viagem ou até mesmo verificar o estado da nossa conta bancária. Atualmente existem 1.8 mil milhões de *websites* ativos, número este que aumenta constantemente, visto que centenas de milhares de novos serviços são criados mensalmente.

A grande maioria dos utilizadores destas aplicações, contudo, não está focada nos aspetos de segurança destes sistemas, visto que as funcionalidades destas é que os atraíram a utilizar as aplicações. Isto leva a que a prioridade das empresas seja oferecer o maior número de funcionalidades no menor espaço de tempo, dado que a variedade de funcionalidades é o que leva os utilizadores a comprar e utilizar os seus serviços. Com isto, as empresas são constantemente confrontadas entre criar novas funcionalidades ou melhorar a segurança das aplicações que desenvolvem, e muitas vezes as empresas colocam no mercado novas aplicações pouco testadas a nível de segurança e que apresentam vulnerabilidades. As startups são empresas que optam pelo desenvolvimento de funcionalidades. Nestas pequenas empresas, gastar tempo e dinheiro extra para criar aplicações sólidas e seguras, adiando as datas previstas de lançamento, pode levar à perda total do negócio, visto que a competição para colocar produtos novos no mercado é alta e a funcionalidade é mais valorizada do que a segurança destes. Além disso, o desenvolvimento das aplicações *web* está cada vez mais acessível, sendo mesmo possível para programadores que não têm grandes conhecimentos técnicos e de programação.

O WordPress é um sistema de gestão de conteúdos que é usado para criar blogues e aplicações *web* de uma forma rápida e intuitiva, sendo assim um sistema apelativo para utilizadores sem, ou com pouca experiência em desenvolvimento de conteúdos *web*. O WordPress é desenvolvido em PHP, uma linguagem de *script* utilizada principalmente para processar os dados do lado do servidor. Estes dois elementos são os mais relevantes no desenvolvimento de aplicações *web*, uma vez que quase 80% usam PHP como linguagem de programação e, de todas estas aplicações PHP, 40% usa o WordPress para gerir os seus conteúdos. Contudo, apesar do seu elevado número de utilizadores, o PHP tem uma fraca validação dos dados, sendo necessário que os programadores tenham boas práticas de programação e que utilizem um conjunto de funções de sanitização para prevenir a inserção de vulnerabilidades no código. Programar em PHP sem gralhas, em termos de segurança de *software*, requer um elevado nível de conhecimentos técnicos da linguagem. Prova disto é que múltiplas vulnerabilidades presentes nos plugins do WordPress estão constantemente a ser descobertas e que afetam todas as aplicações que utilizem esses plugins vulneráveis.

Se combinarmos esta falta de conhecimentos técnicos e más práticas com as limitações da linguagem e as vulnerabilidades existentes, acabamos por ter aplicações *web* com vulnerabilidades de segurança, que estão presentes na Internet à espera de serem exploradas por utilizadores maliciosos. Com a existência destes utilizadores constantemente a atacar as aplicações *web*, as empresas estão cada vez mais a implementar mecanismos de testes em fases tardias no desenvolvimento das aplicações. Apesar deste esforço, o número de vulnerabilidades encontradas continua a aumentar, tornando a segurança destas aplicações uma das suas principais preocupações. Vulnerabilidades de injeção, como *Cross-Site Scripting* (XSS) e injeção de código SQL, continuam a estar muito presentes nestas aplicações, especialmente em aplicações pouco atualizadas. Prova disto é que estas vulnerabilidades estão em terceiro lugar no *OWASP Top 10 2021*, uma lista que contém as vulnerabilidades mais relevantes de momento. O impacto destas vulnerabilidades é enorme e pode resultar na exfiltração de informações sensíveis, perda de dados, ataques de negação de serviço, entre outros. As vulnerabilidades de XSS, apesar de terem um impacto mais moderado, foram encontradas em cerca de dois terços das aplicações, segundo estudos feitos pelo OWASP, revelando a sua grande prevalência na Internet. Apesar de estas vulnerabilidades serem já antigas e muito conhecidas, continuam a existir casos recentes com algumas aplicações que são usadas por milhões de utilizadores.

As atuais abordagens para resolver este problema passam por mecanismos de *fuzzing*, que enviam valores erróneos para as aplicações e detetam vulnerabilidades baseado nas respostas das mesmas. Outras soluções seguem abordagens de análise estática, que analisam o código fonte das aplicações sem a execução do mesmo, e abordagens de execução simbólica que mapeiam as aplicações de forma simbólica para encontrar vulnerabilidades. Por fim, alguma abordagens utilizam *oracles* que são utilizadas para analisar os pedidos feitos às aplicações e definir se esses pedidos são tentativas de ataque. Apesar disto, estas abordagens têm algumas limitações: *Fuzzing* apenas envia valores incorretos para as aplicações e identifica vulnerabilidades baseado nas respostas, não fornecendo nenhuma informação extra de como a vulnerabilidade é explorada nem o código fonte que é vulnerável. O sucesso da análise estática depende do seu conhecimento das classes de vulnerabilidades a testar, assim como dos seus mecanismos para inspecionar o código fonte, que, se forem incorretamente aplicados podem levar a um elevado número de falsos positivos e negativos. Por fim, os *oracles* utilizam *proxies* que apenas identificam se um pedido é malicioso antes da sua execução na aplicação, não tendo conhecimento do fluxo de dados dentro da aplicação nem de como a vulnerabilidade é explorada.

Esta dissertação pretende caracterizar como as vulnerabilidades de *XSS* e *SQL injection* podem ser identificadas no código fonte e como podem ser exploradas. Com esta informação, conseguimos identificá-las no código fonte das aplicações, sem ter de o inspecionar, se correlacionarmos a informação reportada pelos *fuzzers* com a informação recolhida pelo monitor na aplicação *web*, que monitoriza os pedidos recebidos e regista os caminhos de execução dos dados na aplicação. Nós avaliámos e explorámos alguns *fuzzers* com o código fonte disponível, criámos um conjunto de fuzzers para explorar as aplicações e melhorámos um monitor que funciona como *debugger* para recolher os caminhos de execução dos ataques. Os objetivos da tese e a forma de os concretizar são os seguintes.

Primeiramente, é feito um estudo das vulnerabilidades mais comuns nas aplicações *web*, e como é possível explorá-las e identificá-las no código fonte da aplicação. Além disso, é feita uma análise a

soluções de *fuzzing* e injeção de código para aplicações PHP e é identificado como monitorizar estas soluções para recolher os caminhos de execução (*traces*) dos ataques, controlar o seu estado e explorar possíveis vulnerabilidades.

Em segundo lugar, é feita uma melhoria numa extensão do XDebug de forma a poder corretamente identificar os caminhos de execução associados a vulnerabilidades de XSS.

Finalmente, é implementada uma solução, com um conjunto de fuzzers, que injeta código malicioso nas aplicações, recolhe os seus *traces* e identifica se estes estão associados com ataques que conseguiram explorar vulnerabilidades com sucesso, através da correlação da informação dos traces com a informação dos ataques. Se estes estiverem associados a ataques, identifica as vulnerabilidades no código fonte da aplicação, sem o inspecionar, apresentando como resultados as vulnerabilidades encontradas e os ataques que levaram a essa exploração.

A solução implementada contém um conjunto de três *fuzzers* (*W3af*, *Wapiti* e *OWASP ZAP*), os quais se complementam uns aos outros através da partilha dos seus resultados, obtidos na fase de *crawling*. Isto leva a uma melhoria posterior na fase de ataque porque páginas e pedidos que não foram encontrados por um *fuzzer*, mas que foram encontrados por outro, podem ser explorados, usufruindo assim da partilha dos resultados. Desta combinação de fuzzers obtivemos resultados interessantes, tais como a descoberta de vulnerabilidades que não seriam possíveis de identificar caso os fuzzers fossem executados individualmente. Enquanto os ataques decorrem, nós recolhemos os traços de execução gerados durante os ataques feitos pelos scanners, que contêm informação das movimentações dos dados dentro da aplicação *web*, especificamente desde o ponto de entrada do ataque até à chegada de funções sensíveis (funções que podem levar à exploração de vulnerabilidades). Com estes dados que recolhemos, conseguimos identificar os traços de execução que contêm os ataques que foram reportados como bem-sucedidos por parte dos *fuzzers*, confirmar que o valor malicioso utilizado no ataque atingiu uma função sensível e se conseguiu explorar uma vulnerabilidade com sucesso. Finalmente, conseguimos validar a veracidade dos ataques reportados pelos scanners, verificando se os ataques realmente atingiram uma função sensível, assim como apresentar os ataques que foram bem-sucedidos, o traço de execução que levou a essa exploração e a identificação da função vulnerável no código fonte da aplicação. Esta validação é possível devido à informação extra que os traços de execução contêm, nomeadamente os valores exatos que chegaram à chamada de funções e o comportamento posterior da aplicação a estes, sendo então possível identificar se a aplicação tomou um comportamento inesperado e identificar possíveis vulnerabilidades.

A solução foi validada através do teste a três aplicações *web* com vulnerabilidades conhecidas. Os resultados das experiências revelam que a solução é capaz de identificar vulnerabilidades de *SQL Injection* e *XSS*, que não seriam reportadas caso cada *fuzzer* corresse independentemente dos outros. Para além disto, a solução foi capaz de confirmar a veracidade dos ataques reportados pelos *fuzzers*, assim como identificar falsos positivos nos seus relatórios.

Palavras Chave: Vulnerabilidades, Aplicações *Web*, *Ensemble Fuzzing*, Avaliação de ataques, Segurança de Software

Abstract

The vast majority of online services we use nowadays are provided by web applications to the users. The correctness of the source code of these applications is crucial to prevent attackers from exploiting the possible existing vulnerabilities they can contain, leading to severe consequences like the disclosure of sensitive information or the degradation of the availability of the service. Currently, multiple static analysis solutions analyse and detect vulnerabilities in the applications' source code and are used by organisations to test their software. Malicious actors, however, do not usually have access to the source code of these applications and have to plan their attacks based on the information that is made public and fuzzing tools.

This dissertation proposes a solution for the automatic detection of vulnerabilities in web applications written in PHP through an ensemble fuzzing approach and their identification in the source code without accessing it. Using an ensemble fuzzing of open-source web fuzzers, the solution scans the target web applications and collects the application execution traces of the attacks carried by the fuzzers. Furthermore, it correlates the successful attacks reported by the fuzzers with the collected traces, identifying thus the traces in which such attacks happened. Afterwards, the solution inspects the resulting traces to evaluate the truthfulness of the attacks by checking if they successfully reached any sensitive sink (functions susceptible to be vulnerable to malicious inputs), and identifying, for these, the vulnerable code in the application.

The solution was implemented and validated through the testing of three vulnerable web applications using three open-source web application fuzzers for the ensemble. The experimental results demonstrate that it is capable of identifying SQL Injection and XSS vulnerabilities that would be missed if each fuzzer would run without the ensemble. Furthermore, the solution is capable of confirming the attacks reported by the fuzzers, as well as identifying false positives in their reports. Also, for the successful attacks checked, the solution is capable of identifying the vulnerable source code associated with the exploited vulnerabilities.

Keywords: Vulnerabilities, Web Applications, Ensemble Fuzzing, Attack Evaluation, Software Security

Contents

1 Introduction **1**
1.1 Context and Motivation . . . 2
1.2 Objectives . . . 3
1.3 Contributions . . . 3
1.4 Thesis Outline . . . 4

2 Context and Related Work **5**
2.1 Vulnerabilities . . . 5
2.1.1 Injection . . . 5
2.1.2 Cross-Site Scripting . . . 7
2.2 Static Analysis . . . 8
2.3 Fuzzing . . . 11
2.4 Oracles . . . 16

3 Vulnerability Discovery Leveraging Ensemble Fuzzing **19**
3.1 Challenges . . . 19
3.1.1 Fuzzers' exploitation capabilities . . . 19
3.1.2 Fuzzers' code coverage . . . 20
3.1.3 Trace extraction without accessing the source code . . . 20
3.1.4 Identify vulnerabilities without inspecting the source code . . . 20
3.2 Approach Overview . . . 21
3.3 Main Modules . . . 24
3.3.1 Crawlers . . . 24
3.3.2 Request uniformizer . . . 25
3.3.3 Scanners . . . 26
3.3.4 Attack trace collector . . . 27
3.3.5 Attack extractor . . . 28
3.3.6 Interesting traces identifier . . . 28
3.3.7 Vulnerability identifier . . . 30
3.4 Approach's Advantages . . . 31

4 Tool Design and Implementation **33**
4.1 Ensemble Fuzzing 33
4.1.1 Fuzzers' configuration 34
4.1.2 Crawlers 35
4.1.3 Request uniformizer 36
4.1.4 Scanners 37
4.2 Web Application Monitor 38
4.2.1 Xdebug configuration 38
4.2.2 Trace normalization 39
4.2.3 Xdebug modification 40
4.3 Attack Evaluator 47
4.3.1 Attack extractor 47
4.3.2 Interesting traces identifier 49
4.3.3 Vulnerability identifier 53
4.4 Usage Example 56

5 Evaluation **61**
5.1 Web Applications 62
5.1.1 DVWA 62
5.1.2 Mutillidae 62
5.1.3 bWAPP 62
5.2 Ensemble Fuzzing Evaluation 63
5.2.1 Crawler evaluation 63
5.2.2 Scanner evaluation 65
5.3 Trace Analysis Evaluation 69
5.3.1 Metrics 69
5.3.2 Evaluation 70
5.3.3 False false-positive cases 73
5.3.4 Comparison between scanners and trace analysis 74

6 Conclusions **77**
6.1 Future Work 77

References **79**

List of Figures

2.1 Normal and expected authentication in a vulnerable application. 6
2.2 SQL Injection attack example in a vulnerable application. 6
2.3 Reflected XSS Attack Example. 7
2.4 Stored XSS Attack Example. 8

3.1 Architecture of the approach 22

5.1 Number of URLs found by crawlers of each fuzzer when inspecting each web application. 64
5.2 Vulnerability distribution by each fuzzer when attacking each web application. 66

List of Tables

4.1 Characterization of the web application fuzzers evaluated to constitute the ensemble fuzzing. 34

5.1 Number of successfully exploited vulnerabilities in the tested web applications. 65
5.2 An example of XSS of each case. 67
5.3 Confusion matrix for the final results 69
5.4 Number of files, size and traces generated during attack phase by the ensemble fuzzing . 71
5.5 Analysis of the output of the trace analysis tool 71
5.6 Performance test of the solution, for each individual fuzzer, ensemble and global 73
5.7 Comparison of the vulnerability detection in the ensemble version as a whole 74

List of Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CLI	Command-Line Interface
CSRF	Cross Site Request Forgery
CSV	Comma-Separated Values
DB	Database
DOM	Document Object Model
GUI	Graphical User Interface
HSQLDB	Hyper SQL database
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IOA	Indicators Of Attack
IP	Internet Protocol
JSON	JavaScript Object Notation
PHP	Hypertext Preprocessor
SQLi	SQL Injection
SQL	Structured Query Language
URI	Unrestricted File Upload
URL	Uniform Resource Locator
VM	Virtual Machine
XML	eXtensible Markup Language
XSS	Cross-Site Scripting

Chapter 1

Introduction

The Internet is currently part of our day-to-day life, and so applications are built upon it. These applications offer countless services, and they are used to solve multiple needs of our lives, ranging from talking with friends on social media to buying tickets for our next trip, and even checking the status of our bank account. Currently, there are over 1.8 billion websites active in the wild, and every month this number increases by hundreds of thousands as new web services and applications emerge online [6].

The vast majority of the consumers, however, are not mainly focused on the security aspects of these applications, as functionality is what got them to use these in the first place. Thus, this is what companies want to offer the most as it is the main reason that captivates people to buy and use their products. Companies are constantly faced with the decision of choosing between functionality and security for their web services, as the time for security assessment may exceed the planned service launch time, and often they choose to release applications with incomplete security testing. Startups are in this group of companies. In these small companies, spending extra time and money to create good and secure software, delaying the release date may represent the end of the business, as competition is high and functionality is more valued than security. Sohoel et.al [44] studied how startups consider software security in their applications by interviewing employees and testing their applications' code for vulnerabilities. All the tested applications had significant security breaches, and those with the lowest awareness about good practices and well-known vulnerabilities had the most critical security holes. Also, none of the companies had done prior security testing nor used tools for static code analysis and penetration testing and relied on secure code from third parties.

Furthermore, web applications' development is getting more accessible, even for those lacking programming skills. WordPress [1] is an open-source content management system used to create blogs and web applications quickly and intuitively, making it very appealing to inexperienced developers. It is written in PHP, a scripting language used mainly to manage back-end data in web development (i.e., in server-side). These two are the most relevant in web application development. Almost 80% of all server-side websites use PHP as their programming language, and for all websites, almost 40% use WordPress to manage their systems and contents [4][10]. Despite their usage, PHP has poor and limited data val-

idation, requiring the programmers to use the proper functions and have a stable set of good practices. Programming in PHP flawlessly, in terms of software security, is not trivial and requires considerable language knowledge. Also, vulnerabilities in WordPress have been discovered that affect all the websites that use it, and that continue vulnerable if not updated after the patch is released.

Pairing this lack of knowledge and bad practices with the existing vulnerabilities and limitations of the language, we are left with web applications with security flaws that are in the wild, vulnerable and open for anyone willing to exploit their flaws.

1.1 Context and Motivation

Since malicious users are constantly attacking web applications, organisations have been implementing testing mechanisms in later phases of the development process. Although efforts have been employed in this matter, actually the number of discovered vulnerabilities have increased [7], turning the web application security into one of the important concerns of companies.

Injection vulnerabilities, such as Cross-Site Scripting (XSS) and SQL injection, continue to be very prevalent, especially in applications with legacy code. In fact, they are in third place in the OWASP Top 10 2021 list of relevant vulnerabilities in the wild [3]. Its impact is enormous and can result in information disclosure, data loss, denial of service, between others. Despite having a more moderate impact, XSS vulnerabilities were found in around two-thirds of the applications in the OWASP study, revealing their considerable prevalence. Although these vulnerabilities are well known, there are recent cases with some of them in applications used by millions [41].

The current approaches to address this problem rely on fuzzing mechanisms to send erroneous values to the web application and detect vulnerabilities based on the application's behaviour. On the other hand, static analysis solutions analyse the source of the applications without executing its code, and symbolic execution maps the application's state symbolically to find bugs. Finally, oracles can be used to inspect the requests made to the web applications and determine whether they are attempts of attacks. There are, however, multiple limitations in the presented approaches. Fuzzing can only send erroneous values to try the exploitation of the vulnerabilities, only confirming the exploitability based on the web application's responses, not providing information about how the vulnerability was exploited and the vulnerable source code. Static analysis approaches depend on the vulnerability classes knowledge base and the mechanisms used to inspect the source code, which, if incorrectly used, can lead to false positives and negatives. Symbolic execution is usually time-consuming, and many tools have many false positives. Lastly, oracles used as proxies can only identify if a request is malicious by inspecting the requests sent to the application before its processing, not providing thus an in-depth description of how the vulnerabilities are exploited.

This dissertation aims to characterise how XSS and SQL injection vulnerabilities can be discovered in the source code and how they can be exploited. With this information, it may be possible to identify them in the source code of the web applications without inspecting it if we correlate the information

provided by fuzzing tools with the information given by a monitor in the web application that monitors the received requests and their execution paths within the application. We explored and evaluated open-source web application fuzzers, created an ensemble of fuzzers to explore web applications and improved an open-source monitor that acts as a debugger that can collect the execution paths of the attacks.

1.2 Objectives

The main objectives of this dissertation are the following:

- Study some of the most common vulnerabilities present in web applications, how to exploit them, and identify them in the source code of the application. Furthermore, analyse current fuzzing and code injection tools for PHP applications and how to monitor these to collect the application's execution paths (traces) of the attacks, control their state and exploit some possible vulnerabilities;
- Implement a solution that injects malicious code on the web application, collects its traces, and determines if they are associated with the exploitation of vulnerabilities by correlation of information from traces and injected code. If so, identify the vulnerabilities in the source code, without accessing and inspecting the code of the application, presenting as output the vulnerabilities found, and the exploits that lead to that exploitation.

1.3 Contributions

The main contributions of this dissertation are the following:

- A study of web application fuzzers in order to create an ensemble fuzzing which improves the coverage and precision of fuzzing.
- An approach based on an ensemble fuzzing and execution application monitoring for discovering vulnerabilities and identifying them in the source code of the application without accessing to the code.
- An architecture that can be used to explore web applications written in PHP in search of SQL injection and XSS vulnerabilities. It combines open-source web application fuzzers to improve the overall coverage and precision of the fuzzers as a whole, by sharing the results obtained in the web application's crawling and using those to attack the application. Also, the architecture monitors the web application and collects the execution traces of the fuzzer's attacks. Moreover, the architecture has the capability of confirming the attacks reported by the fuzzers by correlating the information of the execution traces with the reported vulnerabilities.
- An improved and extended version of the Xdebug extension able to deal and correctly manage execution paths regarding XSS vulnerabilities.

- A proof of concept through the implementation of a working prototype that:

  1. Ensembles three open-source web application fuzzers and uses the ensemble to share the collected results between them in order to improve the attack phase's results;
  2. Uses a monitor in the server-side of the web application to monitor and collect the execution traces generated during the attacks made to the web application by the ensemble;
  3. Correlates the vulnerabilities reported by the ensemble with the traces collected by the monitor, presenting its report where it confirms the attacks reported by the ensemble and identifies false positives and the source code for the vulnerabilities found. It presents as output the confirmed attacks, false positives, the file and line of the vulnerable source code, the attack used for the exploit and the execution trace.

- An experimental evaluation of the prototype using three vulnerable web applications reveals the tool's viability. Overall, the tool confirmed the reported vulnerabilities, identified new vulnerabilities that would be missed if each fuzzer ran in its standalone version, and reported false positives in the fuzzers' results.

This research led to two publications, a full paper and a fast abstract. The paper *Improving Web Application Vulnerability Detection Leveraging Ensemble Fuzzing*, in the International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'21) [21], and the fast-abstract *Finding Web Application Vulnerabilities with an Ensemble Fuzzing*, in the 51st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'21) [20].

1.4 Thesis Outline

This dissertation is organised as follows:

Chapter 2 analyses relevant terms and concepts, provides fundamental context for the work and presents some related work.

Chapter 3 presents the study of the web application fuzzers made to define the ensemble fuzzing, and the approach overview of the architecture we propose to cope with the objectives of this dissertation.

Chapter 4 describes the implementation of the proposed architecture and the decisions we made to its implementation.

Chapter 5 presents the evaluation of the solution with an ensemble of three fuzzers and three different web applications.

Chapter 6 provides the conclusions of the developed research and discusses possible future work.

Chapter 2

Context and Related Work

This chapter presents the necessary concepts that support the goal of this dissertation and gives an overview of the related work of the problem we address. Section 2.1 provides details of vulnerabilities in general and goes into detail of the vulnerabilities that we explored in our implementation. Section 2.2 provides an overview of some techniques and problems of static analysis. Section 2.3 presents different fuzzing techniques and tools. Lastly, Section 2.4 presents related work regarding oracles.

2.1 Vulnerabilities

There are multiple definitions to what a software vulnerability is, but it can be described as a weakness in a system, usually as the result of bugs introduced in the software development phases, that can be accidentally triggered or intentionally exploited by an attacker, resulting in unexpected events that generally violate the system's security policies [12][32][13].

PHP is a programming language that, from the security standpoint, is poorly designed. It typically yields a large attack surface and bears inconsistently designed functions with often surprising side effects, all of which a programmer must be aware of and keep in mind while developing a PHP application [33]. This reflects the increased number of vulnerabilities found in PHP applications. For instance, compared to 2016, 2017 saw a 400% increase of reported vulnerabilities in the top four most popular PHP content management systems [15].

2.1.1 Injection

Injection vulnerabilities are the result of a lack of proper input validation and sanitization of a program. It allows adversaries to provide untrusted input, meticulously created so that they are wrongly processed by a code interpreter, usually sent though some data submission to the web application.

They were at the top of the 2017 OWASP Top 10 report [14] and are in third place in the report

from 2021 [3]. This is due to it's easy detectability and exploitability, common prevalence among web applications and high impact, as it can lead to data exposure, denial of service, loss of data integrity or even a full system takeover.

SQL Injection is probably the most known type of injection attack. Figure 2.1 illustrates the normal case of a successful authentication of a user in a web application through a login form. Figure 2.2 shows a simple example of this kind of attack, where a malicious actor logs in as the user "Admin" without knowing his password. To achieve this, he sends, as the password, the forged string `' or '1'='1`. Since this database has poor user input sanitization, it allows for the attacker to insert a tautology in the query to be evaluated by the database. By doing this, when the query is processed to assess if the given password for the user "Admin" is correct, it will evaluate the expression `password=''` **or** `'1'='1'`, which, because of the tautology, will always be successful. As result, the actor successfully logs in as the user `Admin`.

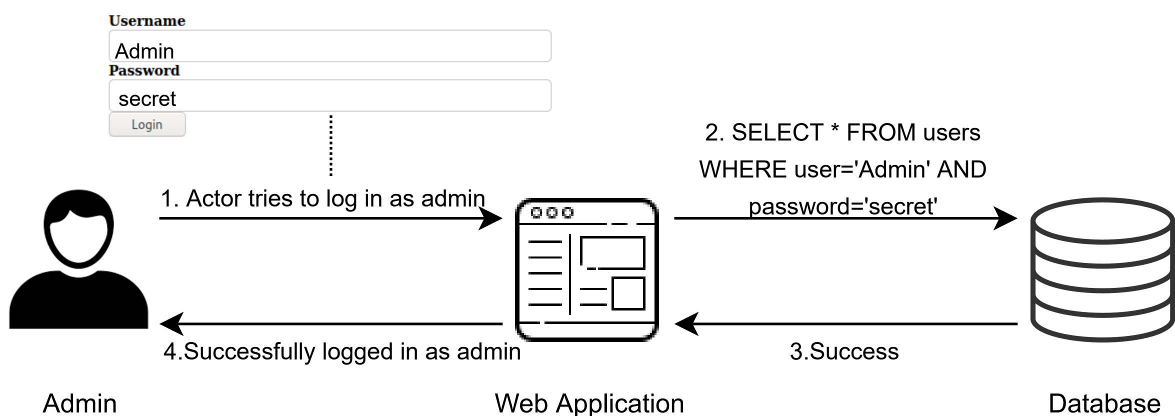


Figure 2.1: Normal and expected authentication in a vulnerable application.

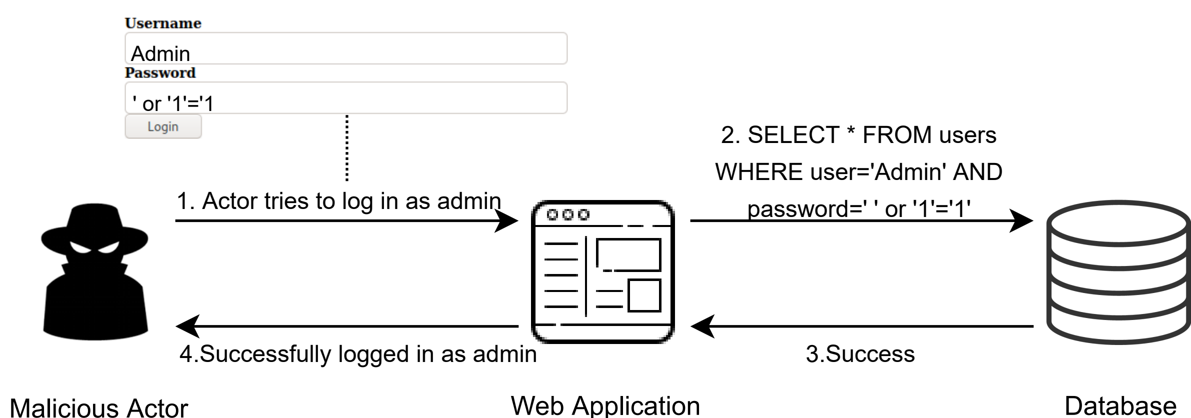


Figure 2.2: SQL Injection attack example in a vulnerable application.

To avoid creating this kind of vulnerabilities, user provided input should never be trusted and functions and features that are known to be secure against this attacks should be used. Prepared statements are templates for queries, usually in SQL database systems, that use placeholders to receive the user input and assemble and sanitize the data upon execution, making it a possible solution that could be applied in the example above.

2.1.2 Cross-Site Scripting

Cross-Site Scripting (XSS) is a code injection attack that allows an adversary to inject malicious scripts in a website. There are three types of XSS attacks:

1. **Reflected XSS**: Reflected XSS attacks occur when an attacker is able to reflect a malicious script of a web application into the browser. Figure 2.3 illustrates a possible Reflected XSS attack. A malicious actor prepares the attack by creating its own web server, that will be used as endpoint to send the victim's sensitive information, and a script that sends the session information to its web server. The first step is to send a malicious link to a targeted victim (especially effective in phishing and spear-phishing attacks). The victim clicks the link and sends the request to the vulnerable web application, that will return to the victim a web page containing the script. This script will then execute in the victim's browser and send a request containing his session information to the malicious actor's web server. The attacker can then access this data and impersonate the victim by using his session information, making it possible to execute functions as the user.

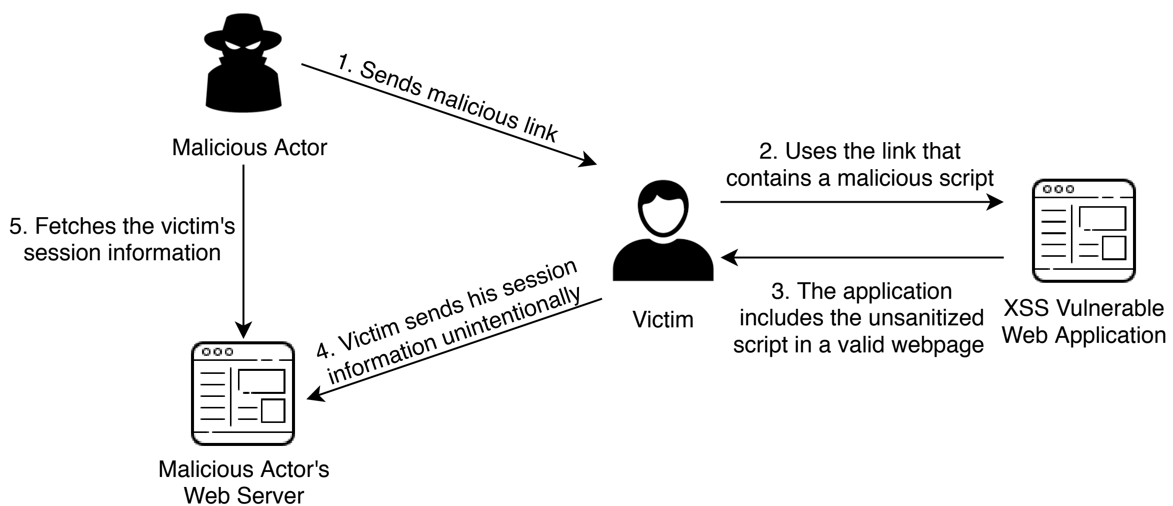


Figure 2.3: Reflected XSS Attack Example.

2. **Stored XSS**: Stored XSS attacks occur when an attacker is able to store malicious user input that is not encoded nor validated by the web application. Having the script stored, later accesses by users

will receive and execute the malicious code, triggering, for example, an XSS attack. Figure 2.4 presents an example of a Stored XSS Attack. This attack is very similar to the one presented in the Reflected attack, however, here the malicious actor has no direct contact with the victim. Instead, he is able to store a script in the web application that will affect all the users that access that zone.

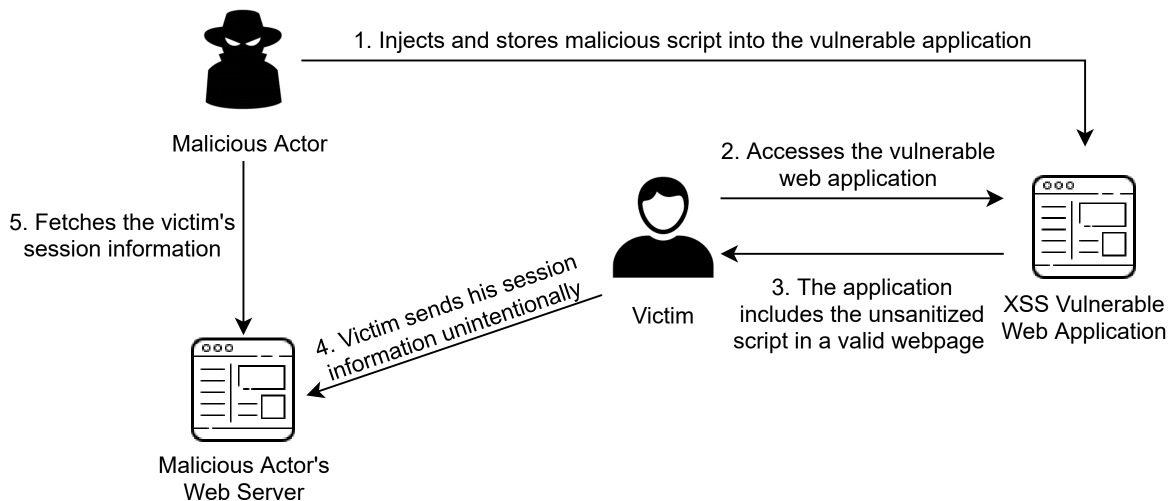


Figure 2.4: Stored XSS Attack Example.

3. **DOM Based XSS**: DOM Based XSS attacks are possible when the application writes data to the Document Object Model (DOM) in the victim's browser without a proper sanitization. The attack is similar to the Reflected XSS attack, however, in this case, the web page from the application responses is static and the client side code executes abnormally due to the malicious input sent in the link. In this type of attack, it is even possible that the malicious payload is never sent to the website's server.

2.2 Static Analysis

With the objective of finding code implementation bugs, static analysis tools analyse the program statically (without executing it) through its source code or compiled form. They have the ability to check the code in an unbiased and consistent way, often being able to identify the root cause of security problems. These tools are often used because they can review a large amount of code. This is especially important when a new variety of attacks are discovered, and it is necessary to analyse legacy code.

One form of static analysis is manual auditing, where the source code is examined by a security expert auditor. Although effective in some cases, it becomes very time consuming and usually infeasible when the size of the source code increases. Static analysis tools are much faster and can incorporate the security

knowledge required, allowing its use by auditors that do not need the same level of security expertise. A good tool must allow its output to be understandable to normal developers who might lack good practices in software security.

The static analysis is, however, a computationally undecidable problem. Rice's theorem [40] says that any question that can be applied to a program can be reduced to the halting problem. With this, static analysis cannot perfectly determine any nontrivial property of a general program, making it an undecidable problem in the worst case. This forces static analysis tools to make approximations that lead to non-perfect results. These tools usually carry a trade-off between false positives (report bugs that do not exist) and false negatives (do not report bugs that exist). While a big rate of false negatives is more dangerous because it leads to a false sense of security, a big rate of false positives tends to make people stop using the tool, as it causes grief in the analysts that have to analyse the reported vulnerabilities (that do not exist) and the resources used will leave the sensation of wasted time. Nonetheless, despite having some imperfections, it does not prevent static analysis tools from having value; The important thing is to provide some useful output.

One of the simplest approaches to static analysis is the Unix command *grep*, which searches for a given string in some file. While good for searching for insecure functions, it does not interpret the code it scans, meaning that it is not able to distinguish comments, string literals, declarations and function calls. Another basic approach is the lexical analysis, in which a preprocessing and tokenization of source files is performed, and the resulting tokens are compared with some known vulnerable constructs. Even though this approach is better than the string matcher approach, it still produces a hefty number of false positives, due to the lack of knowledge of the target's semantics.

In order to increase precision and achieve better results than the above-described approaches, a tool may leverage on more compiler technology by using abstract syntax trees (AST), which take into account the semantics of the evaluated program. The scope of the analysis is important in this case, as it determines the amount of context considered; More context reduces false positives but boosts the computation needed. We have three types of analysis based on the scope and amount of context: Local analysis, that does not consider interactions between functions and studies one function at the time; Module-level analysis, that takes into account the relationships between functions and class properties within the same modules, but not calls between different modules; Finally, global analysis, that examines the entire program and all relationships between functions.

Data flow analysis is a technique that computes the information about the flow of data for each point of the program being analysed, and uses it to approximate the desired properties of the run time behaviour. The program's control flow graph (CFG), built using the program AST, is used to calculate the parts of the program to which a value assigned to a variable might propagate. The most common form of data-flow analysis used in the security context is taint analysis, which has the objective of identifying the values of the program that could be potentially controlled by an attacker. Program data is either defined as tainted or untainted, and uses a variety of rules to determine its type. It is essential to identify the entry points of the program where tainted data enters the system, sink locations that should not receive tainted

data and cleanse zones where tainted data is sanitised and becomes untainted. This tracking of tainted data is relevant to identify security vulnerabilities since many of these problems result from trusting untrustworthy input [24][25].

Opposingly to static analysis, dynamic analysis evaluates the system while it is running, providing the applications with test inputs to cover all the possible outputs in the testing application. Static and dynamic analysis have, however, shortcomings that can result in a large number of false positives and false negatives. To overcome this, and with the intention of reducing these numbers in both static and dynamic analysis methodologies used in PHP web applications, Zhao and Gong [50] proposed a framework that combines both static and dynamic analysis. They exercised static analysis to get the special paths, parameters and others, and constructed a dynamic test set that was used to detect real dynamics of vulnerabilities. By doing this, they made use of the advantages of both static and dynamic analysis, and significantly improved the dynamic analysis with the data collected by the static analysis, improving the efficiency and detection of vulnerabilities, and thus reducing the number of false positives.

Some examples of studies that used static analysis to identify vulnerabilities are presented below.

Dahse and Holz [27] presented RIPS, a tool that builds control flow graphs and creates block and functions summaries by simulating the data flow for each basic block, which allows to conduct a precise taint analysis. In doing so, the authors discovered previously unknown flaws in the web applications osCommerce, HotCRP, and phpBB2. In their follow up work [26], they detected second-order vulnerabilities, e.g., persistent Cross-Site Scripting, and identified more than 150 vulnerabilities in six different web applications.

Since modern web applications use multiple dynamic features that pure static analysis solutions may not be able to study, Alhuzali et al. introduced NAVEX [16], an automatic exploit generation framework that efficiently generates exploits. Their approach combined both dynamic and static analysis techniques to identify the paths from sources to the vulnerable sensitive sinks while considering sanitisation filters, taking into account multiple vulnerability classes, including SQLi and XSS. They modelled the applications using symbolic execution and performed a dynamic analysis of the applications using a web crawler and a concolic executioner, discovering paths that may lead an attacker to a vulnerable sensitive sink, and generating the exploits that may trigger incorrect behaviours in the applications.

Medeiros et al. [37] presented a solution that combines static analysis with data mining to detect vulnerabilities in the source code of web applications written in PHP. They used taint analysis to find candidate vulnerabilities and data mining to predict possible false positives in the chosen candidates. Furthermore, they proposed an automatic code correction approach that inserts fixes in the application's source code. They applied their solution in WAP (Web Application Protection) [36] and got a low false-positive rate in their results.

Taking into account the differences in the results provided by distinct static analysis tools, Nunes et al. [39] proposed a benchmark that compares static analysis tools in terms of their capability of detecting XSS and SQLi vulnerabilities. They tested five free PHP static analysis tools (RIPS, Pixy, phpSAFE, WAP, and WeVerca) in 134 WordPress plugins. The results varied according to the deployment scenario

and the vulnerability classes being explored.

2.3 Fuzzing

Fuzzing is an automatic or semiautomatic software testing technique to find vulnerabilities, which involves providing invalid and unexpected inputs to a target and monitoring for exceptions such as crashes, memory leaks, or information disclosure [45].

Fuzzers can be divided based on three categories:

- Input format knowledge;
- Target application structure knowledge;
- Method of input generation.

If the fuzzer has no awareness about the format of the input data, it is considered a dumb fuzzer. Without this knowledge, the data generated for fuzzing is considered random, as the probability of the data being in the correct format is very low. On the other hand, in smart fuzzing, commonly referred to as structure-aware fuzzing, the fuzzer has the capability of generating data semi-randomly, as it is aware of the expected input data format. While being better than dumb fuzzers in some cases, the process of creating this input model that allows the construction of inputs in the expected format may be difficult due to proprietary issues, complexity or simply because they are unknown.

In terms of the knowledge of the application structure, a fuzzer can be one of the following types:

- **White-Box**: In this case, the fuzzer has access to the application's internal structure, processes and functionalities, and it is possible to apply symbolic execution on the application under test, gathering constraints from conditional branches. It has the main advantage of having a greater coverage when compared with other techniques. It is, however, a slower process and has problems like the path explosion that derives from the exponential growth of the possible paths to explore.
- **Black-Box**: The fuzzer does not have knowledge about the internal structure of the application. With this, it is harder to verify the branches covered by the execution, as the fuzzer only has as information the responses given by the application.
- **Grey-Box**: In conjugation with white-box and black-box, the fuzzer leverages on lightweight code instrumentation instead of program analysis to obtain information about the system. Even though this brings an additional performance overhead, this knowledge grants increased code coverage.

Lastly, there are two approaches for "how" the data used for fuzzing is created: generation and mutation. In generation-based fuzzing, the fuzzer generates the input data without relying on previous inputs

or existing seeds, usually learning models of input and generating new inputs based on the learned models. Contrarily, mutation-based fuzzing modifies input based on defined patterns and existing seeds [18].

Regarding black-box fuzzing, some of the online-available and free web application fuzzers are presented below:

- **Burp Suite** [2] is the most used security testing software for web applications. Created by the company PortSwigger and written in Java, it is known for its paid version with advanced tools and is continually updated. They also have a free community version that has minimal features available.
- **OWASP ZAP** [11] is an open-source web application scanner and is an active project of the OWASP community. Written in Java and ranked 1 in the 2013 Top Security Tools [46], voted by ToolsWatch.org readers, ZAP is still a reference for those new to web application security, as well as professional penetration testers.
- **Skipfish** [5] is a web application security reconnaissance tool that crawls and probes the application based on dictionaries. It is written in C, has a minimal CPU footprint, is easy to use and has a low false-positive rate. Despite being outdated, it is still distributed in the recent Kali Linux Operating System versions, a widespread distribution for penetration testers.
- **Wapiti** [9] is an open-source web application vulnerability scanner written in Python 3 that has very recent updates. Despite not having a Graphical User Interface (GUI), it is easy to use and has detailed documentation.
- **XSStrike** [42] is a Python web application scanner specialized in Cross-Site Scripting vulnerabilities. Its injection mechanism is different from other scanners because it analyses the responses with multiple parsers and crafts payloads guaranteed to work in that context. It only has a Command-Line Interface (CLI), but it is simple, and there is a wiki available with tutorials.
- **w3af** [8] is a Web Application Attack and Audit Framework developed in Python that has both command line and graphical interfaces. It is a project from 2009, started by Andres Riancho, an application and cloud security expert, currently Head of Information Security at Wildlife Studios, that has grown and has been a reference in web application fuzzing, despite not being currently actively developed.

Below we present some of the related work that uses fuzzing approaches to detect vulnerabilities.

KameleonFuzz [31] is a black-box XSS fuzzer for web applications that targets reflected and stored XSS. It is intended to mimic a human attacker by prioritising the most promising inputs to fuzz. KameleonFuzz uses the Evolutionary Fuzzing method, which uses machine learning algorithms to evolve the values used to fuzz web applications by using the feedback from the test cases. In a first phase, it generates a fuzzed value according to an attack grammar, manually constructed using knowledge of HTML

grammar, string transformations and known attack vectors for the reflected parameter. In the second phase, it obtains information about the context of the XSS reflection, determining if the fuzzed value triggered an XSS vulnerability and evolving the values based on how close they were to exploiting a vulnerability. In their evaluation, KameleonFuzz detected most of the XSS detected by other scanners, several missed by others and three 0-day XSS vulnerabilities.

Demetrio et al. created WAF-A-Mole [28], a tool that models the presence of an adversary. From the class of *Guided Mutational Fuzz Testing* approach, they started from a failing test that got repeatedly mutated with the use of mutation operators. The result tests were then executed, compared to some metric and ordered accordingly, repeating the process until a successful test was found. They implemented the payload pool as a priority queue, prioritizing the payloads according to the confidence value returned by the classification algorithm. The main idea was that the head element of the queue (the payload with the lowest confidence score) was extracted, mutated, classified and enqueued again. With their results, the authors demonstrated that web applications firewalls based on machine learning are exposed to a risk of being bypassed.

Vimpari studied [47] some of the open-source fuzzers available at the time, analysing the differences between them from the perspective of someone with basic knowledge in software testing. He ended up selecting six fuzzers: Radamsa, MiniFuzz, Burp Suite, JBroFuzz, w3af and OWASP ZAP, the last four being web application fuzzers.

The criteria used in the comparison were the following:

- Ease of use, like how easy it is to install the tool and the environmental requirements;
- Practical features, like the built-in automation and target monitoring support;
- Fuzzer intelligence, like the data generation method and the use of models and heuristics.

The author concluded that the best fuzzer depends on the use case intended. However, he presented the following advantages and disadvantages for the studied web application fuzzers:

- Advantages:
  - OWASP ZAP: Good support for automation and a large collections of plugins for different types of data;
  - W3af: Large collections of plugins for different data types;
- Disadvantages:
  - Burp Suite: Limited free edition;
  - JBroFuzz: No available support;
  - OWASP ZAP: No beginner-friendly documentation;
  - W3af: No Windows support and tool distributed as source code.

Challenged by the complexity of real-world applications, Chen et al. [23] studied the performance of an ensemble fuzzing approach. They started by reviewing and selecting base state-of-the-art fuzzers based on their input generation strategies, seed selection and mutation processes, and the diversity of coverage information granularity. Secondly, they implemented a system global asynchronous and local synchronous that periodically synchronizes fuzzers attacking the same application, sharing between these fuzzers interesting seeds that cover new paths or generate new crashes. To test their solution, they applied it to several real-world applications and discovered 60 new vulnerabilities, of which 44 were registered as new CVEs.

Taking into account the struggles in creating a methodology and standard tools for the evaluation of fuzzers, FuzzBench [38], an open-source platform researched and created by Google, offers free, fast and reliable and reproducible services for evaluating fuzzers. Any user can request a new experiment evaluation for free, and the tool uses Google's compute resources to benchmark the capabilities of the testing fuzzer. They benchmarked the fuzzers AFL, AFLFast AFL++, AFLSmart, Eclipser, Entropic, FairFuzz, Honggfuzz, libFuzzer, MOpt-AFL, and lafintel, in their default benchmark set, containing 22 different open-source projects. In their evaluation, the fuzzer AFL++ came out the best, followed by Honggfuzz, Entropic, and Eclipser. Since the service is unbiased, free and largely scalable, FuzzBench has been used by dozens of researchers from both academia and industry backgrounds.

One of the biggest challenges in black-box tools is to determine the interactions that can change the application's state. For example, sending the same requests in a different order can result in different paths explored. Without taking this into account, a big portion of the application could be missed, and multiple vulnerabilities overlooked. To address this problem, Doupé et al. [29] proposed the use of a symbolic Mealy machine, a finite-state machine that calculates the output and the next state given some input and the current state, to incrementally model the web application. Their solution navigates through the web application, making requests similar to those sent previously and inferring if the application's state has changed, focusing on the idea that if two requests are similar and the response is different, then the state must have changed. They prioritised requests that were less likely to cause a state transition and evaluated, grouped and collapsed similar states to minimise the total number of states and avoid repetitions. Their solution was then compared with multiple web application scanners, including w3af, and achieved better coverage and vulnerability detection in all tests, including unique vulnerabilities.

Another approach to tackle the changes in the application's state was followed by Eriksson et al., which created Black Widow [34], a black-box scanner for finding XSS vulnerabilities that uses an inter-state dependency analysis, focused mainly on identifying store XSS vulnerabilities. Because web application scanners are not usually able to discover inter-page data dependencies and miss vulnerabilities that can only be triggered after certain data flows in the application, the tool identifies input fields, injects malicious tainted values, and later checks if the inputs appear in the HTML pages. By keeping track of the injected tainted values, it allows for a better understanding of the dependencies between the different pages, allowing the detection of more sophisticated XSS vulnerabilities. Their tool uses a crawler that is able to explore both the static structure of webpages (e.g., forms), as well as JavaScript events (e.g., on-

MouseClick), following first a breadth-first search in its traversing to gain an overview of the application before diving into specifics. Furthermore, it logs the sequences used in the crawled pages, enabling the retracing of these sequences. Differently from other scanners, Black Widow does not have false positives due to their dynamic verification of the code injections and has an increased code coverage compared to other solutions, as well as in the discovery of vulnerabilities.

Sargsyan et al. [43] presented a tool that follows a different directed fuzzing approach with the main objective of executing interesting code fragments as quickly as possible. Given a list of addresses to be executed, they identified the paths from the application's entry points to the destination addresses, pinpointing blocks that can modify data and influence the execution. With this, they evaluated the fuzzer generated input and focused on those with the highest score; Consequently, the generation of the fuzzer input speed, that explores special code regions, was faster. The authors compared their approach with AFL and detected multiple crashes missed by it, in the presented execution time.

Similar to the above web application fuzzers, Đurić created Web Application Penetration Testing Tool (WAPTT) [51], a black-box testing framework that simulates attacks against web applications. It automatically crawls the web application (complementing with a manual navigation to explore all the scenarios), identifies and extracts the entry points by parsing the visited pages, fuzzes the entry points with generated payloads that violate the constraints of the parameters, and finally, analyses the HTTP responses of the web application in search for vulnerabilities. By the end, it generates a report with the vulnerabilities found. It has the same approach as the common web application fuzzers. However, it was able to detect more vulnerabilities (at the time), and has a high modularity, meaning the user may easily extend WAPTT's functionalities.

The DOM is responsible for displaying HTML documents in an interactive window to the users, and its engine web browsers are a common attack surface for attackers. In order to detect vulnerabilities in the DOM, Xu et al. proposed FreeDom [49], a DOM fuzzer framework that works with both generative and coverage guided mutation modes. FreeDom relies on a context-aware intermediate representation to describe HTML documents with proper data dependencies. To detect the vulnerabilities, it generates a new document, fuzzes the elements of the documents, merges the documents together and executes the generated code in the web browsers.

Another proposed solution to detect vulnerabilities in PHP applications is BaZINGA [19], a white-box fuzzer that combines static analysis, concolic testing and fuzzing. It injects payloads in the application and statically analyses the code from the application's execution, solving the constraint paths and generating new inputs to be used in the fuzzing process, in an iterative way. By doing this, it can generate sufficient payloads that can cover 100% of the application's code, according to the presented results. Compared to our work, BaZINGA uses static analysis approaches that inspect the source code, generating new fuzzing payloads based on that analysis, achieving excellent code coverage. On the other hand, our solution does not inspect the source code and instead inspects the traces generated in the fuzzing process, which contain information about the exact values that reached the functions. Since it contains such information, dynamic aspects of the application are considered in our case, which may lead to an increased rate of

false positives in static analysis solutions.

2.4 Oracles

Regarding web applications' security context, an oracle is a module that is able to detect attacks based on the data that is passed to it. Oracles can, for example, act as a proxy, intercepting and analysing the traffic that is entering the application and classifying it as potentially dangerous. 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 approaches, like performing a labelling action as an attack or non-attack, to more complex approaches, that provide additional context about the action that is being attempted.

In a vast majority of the current open-source web application fuzzers, the oracle module works by comparing the injected inputs with the resulting output obtained from the web server response. They have available information about the behaviour of malicious inputs and their expected output and are able to assess if the input is malicious.

Server-side oracles can have access to information about the code execution in the web applications execution traces, also known as function traces (or simply traces), are a way of logging a program's execution. An execution trace may contain the recording of the whole execution or only part of it, although that depends on the feature set of the tracer. PHP's most popular tracer is Xdebug[1], which is an extension of the PHP interpreter. Xdebug can be configured to record quite some information, including which functions were called, what arguments they were supplied with and what values the functions returned. It also records the call tree, which shows the relation between functions calls.

Finally, we present some of the related work that apply different types of oracles in their search for vulnerabilities.

Ceccato et al. proposed SOFIA [22], a security oracle for SQLi attacks detection that has no information about known attacks nor the source code of the applications. SOFIA has a training phase where only benign statements are fed to the system, which proceeds to apply machine learning mechanisms, namely tree parsing, pruning, classification and clustering. With these steps, the authors created a safe model that was then used in the testing phase to compare requests and classify as attacks everything that did not conform with this model. They compared the structure instead of the data, which enabled them to achieve a low false-positive rate and a high recall. In their results, the authors achieved a very high accuracy when classifying legitimate and attack statements, outperforming other tools with similar approaches, and a fast classification time.

Regarding the authentication and authorization flaws involving web application sessions and cookies, Drakonakis et al. developed an automated auditing framework [30] that audits web applications in a

[1] https://xdebug.org/

black-box manner, evaluating their susceptibility to cookie-hijacking attacks and security mechanisms to prevent it. They implemented multiple oracles to identify the correct signup and login in the tested web applications in order to successfully audit the generated session cookies. For instance, they checked for cookie flags like *secure* and *httpOnly*, which, if not present, may allow an attacker to perform eavesdrop and highjacking attacks. Additionally, they detected the exposure of sensitive personal data using the collected vulnerable cookies, validating the danger of the application's security misconfiguration. Finally, they developed Xdriver, a fault-tolerant custom browser-automation tool is built on top of Selenium that efficiently handles unexpected events and errors, allowing for the tool to continuously perform without restarting the whole process.

FUSE [35] is a tool designed to identify unlimited file upload vulnerabilities in PHP applications. It generates upload requests using 13 mutation operations that transform the seed files. The main goal is to successfully upload a file, bypassing the filters applied by the applications, while maintaining the file executable on the server side. It is a black-box assessment, and the successful upload of an executable file implies a vulnerability in the application.

Antunes proposed a solution [17] that automatically detects and removes vulnerabilities in PHP web applications. It monitors the requests received and the output of the application, searching for malicious payloads. With the detected payloads, it uses the payloads to scan the application's source code in search of vulnerabilities, concluding with the correction of those. Antunes 's solution differs from ours since it implements an interceptor between the fuzzer and the application, collecting both the requests and responses, while we use the fuzzers' capabilities to get access to those. Also, his solution identifies malicious payloads and identifies the vulnerabilities in the source code by using directed static analysis and code fragments approaches. On the other hand, we inspect the generated execution traces that contain further information about how the malicious payloads reached the sensitive functions and the exact values that reached such functions.

Chapter 3

Vulnerability Discovery Leveraging Ensemble Fuzzing

This chapter presents our proposal for improving the detection and identification of web vulnerabilities by leveraging an ensemble fuzzing and monitoring its output.

Section 3.1 presents the challenges that our solution aims to resolve. Section 3.2 provides an overall view of the solution we propose. Section 3.3 presents more details regarding each of the main modules that compose the solution. Lastly, Section 3.4 presents the advantages of our solution.

3.1 Challenges

This section presents the four challenges we face when using web application fuzzers to exploit and identify vulnerabilities in the source code of web applications, and that we attempt to resolve with our solution.

3.1.1 Fuzzers' exploitation capabilities

The web application fuzzers comprise two components - **crawler** and **scanner**. The former inspects the web application's attack surface under test to extract the URLs and their entry points (web requests), whereas the latter performs attacks over these URLs, by injecting malformed inputs through their entry points, in attempting to exploit some existing vulnerability. However, there is no certainty that the crawler extracts all the valid URLs the application contains, i.e., if it can extract all the entry points from the attack surface and compose valid URLs with them. Contrarily, for the scanner, there is no guarantee that it can exploit the possible vulnerabilities contained in the application, i.e., if it can exercise the URLs with the correct payload (i.e., malcrafted inputs) capable of exploiting the vulnerabilities.

In an attempt to reduce these limitations, we propose an ensemble fuzzing approach that allows us to combine multiple fuzzers and share their crawlers' results with one another, which the scanners will use. Although this process does not increase each fuzzer's individual capabilities, it allows all the fuzzers to complement each other, thus increasing the overall performance of the fuzzers.

3.1.2 Fuzzers' code coverage

Another challenge is the problem of the interactions that can change the application's state, presented previously [29]. Similar interactions with the web application can possibly get different results. For instance, using the same requests with distinct inputs or sending them in a different order can result in different executed paths. Hence, without considering the possibility that identical requests can produce different results, the code coverage is worse, and vulnerabilities could be missed.

The share of the fuzzers' findings in our ensemble approach allows for an increase in the code coverage (because the fuzzers have different crawling capabilities and more fuzzers will naturally find more endpoints) and vulnerabilities exploited by increasing the probability of exploitation by another fuzzer. Take for instance a fuzzer whose crawler is not able to detect the startpoint related to an endpoint that is vulnerable to attacks and that this startpoint is detected by another fuzzer that is not able to exploit it; By sharing the crawler's information between them, the first fuzzer may be able to exploit the vulnerability and thus increase the final number of vulnerabilities found.

3.1.3 Trace extraction without accessing the source code

An additional challenge is the capability of extracting the web application execution paths (traces for short) generated from the fuzzers' attacks without accessing the source code of the web application directly. To surpass this challenge, we propose a web application monitor with debugging capabilities to record the traces and the function calls resultant from the interactions of the fuzzers' requests with the target web application.

3.1.4 Identify vulnerabilities without inspecting the source code

The final challenge is to identify which of the collected traces are associated with the exploited vulnerabilities and, therefore, identify the vulnerable source code without inspecting the source code of the web application. Unlike static analysis approaches that actively inspect the source code of the applications, we propose a solution that does not require the analysis of the applications' source code and instead inspects the execution traces generated with the users' interactions with the web application, and correlates them with the outcomes of the scanners.

3.2 Approach Overview

This section presents our proposed approach and the modules it depends on. The approach aims to detect and identify vulnerabilities by combining different web application fuzzers and sharing the fuzzing results between them to achieve a more significant number of found vulnerabilities and an improved code coverage of the application. Furthermore, we introduce our technique of collecting the web applications' execution paths (traces) of the requests made by the fuzzers and provide an overview of our method to visualize the vulnerabilities in the source code of the web application through the identification of the successful attacks in the collected traces and the confirmation of the successful exploitation of the reported vulnerabilities. Here we use the information collected relying on the web application's responses, the information collected during the fuzzing attacks, and the web application execution.

Figure 3.1 presents the architecture of the approach, composed of three main modules - *Ensemble Fuzzing*, *Web Application Monitor*, and *Attack Evaluator* -, and the interaction within and between them.

1. **Ensemble Fuzzing**

   This module is responsible for synchronizing multiple web application fuzzers, executing them, and sharing their collected information between all the fuzzers. Also, it ensures that all fuzzers perform the requests correctly to the target web application and that the requests are being correctly collected and distributed between the fuzzers. It has the following components:

   - **Fuzzers**: Test the web applications in search of vulnerabilities. The fuzzers can be subdivided into three sub-modules:
     - **Crawlers**: Recursively crawl the attack surface of the web application and collect the URLs to explore by the Scanner with the parameters and values they used in the requests they performed to obtain the URLs;
     - **Scanners**: Perform attacks by recursively exercising the URLs with different values (malicious inputs) and inspect the responses of the web application in search of vulnerabilities (successful attacks);
     - **Fuzzers' configurations**: The configuration settings used by the crawlers and the scanners, usually related to authentication credentials, URLs to ignore (e.g., the logout URL) and the vulnerability classes to explore;
   - **Requests' DB**: The database where the crawlers store the information while crawling the application, namely the URLs, HTTP methods, parameters and values used in the requests, and the web application's responses. Each fuzzer has its database that is independent of all the other Fuzzers;
   - **Request Uniformizer**: Uniformizes the format of the requests among the different crawlers, i.e., adapts the format of the requests of one crawler to the other crawlers, allowing the scanner

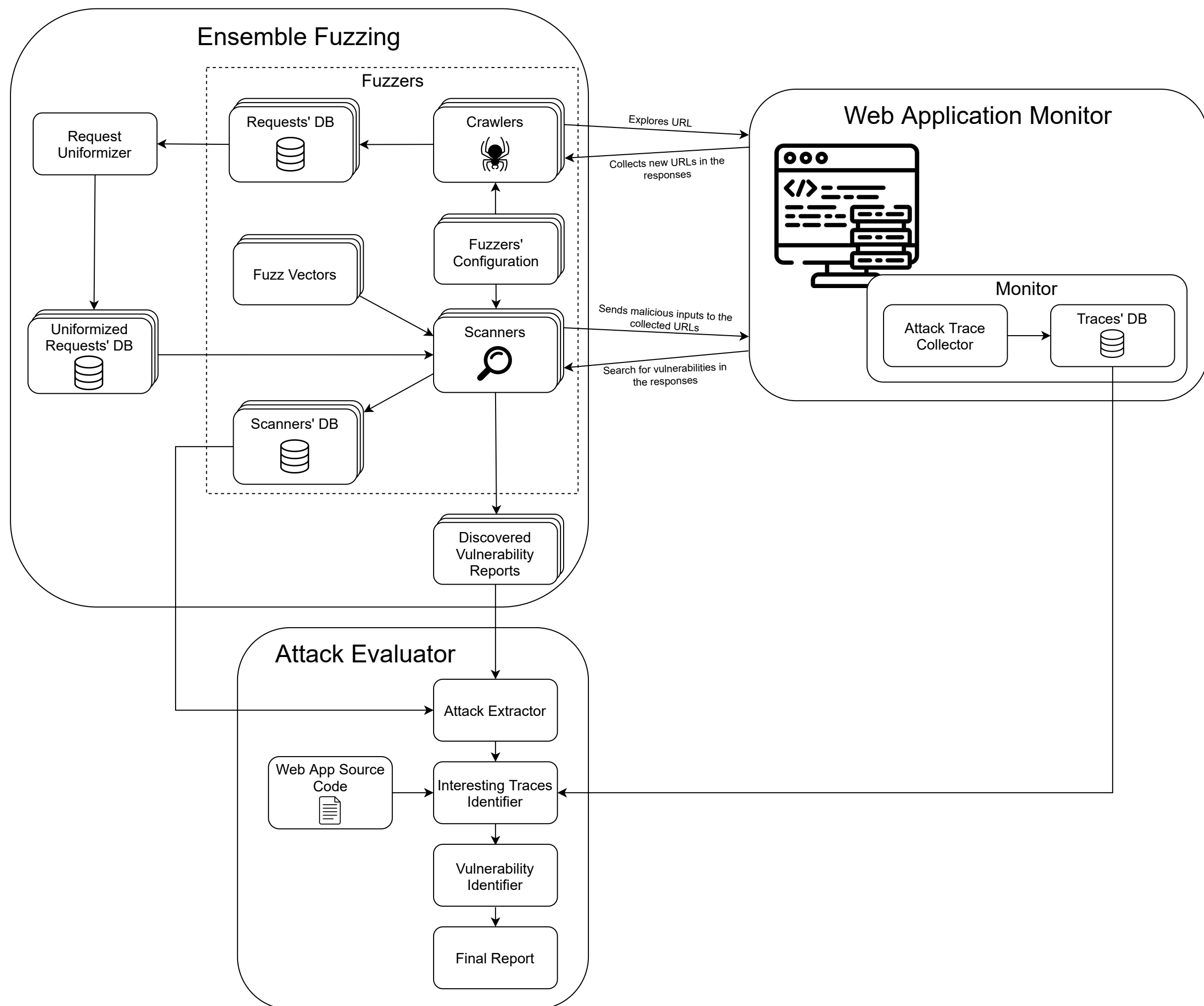


Figure 3.1: Architecture of the approach

of the latter to use the requests retrieved by the former. Also, duplicated requests are identified and removed, generating thus unique requests;

- **Uniformized Requests' DB**: The database containing the unique requests represented under the request format for each fuzzer;
- **Fuzz Vectors**: Attack signatures for the chosen classes of vulnerabilities, which have a statistically higher probability of triggering a bug. Each scanner uses its own fuzz vector;
- **Discovered Vulnerability Reports**: The reports generated by the scanners after testing (attacking) the application, containing information of the successful attacks, more precisely the type of vulnerabilities exploited, the URL, parameters, malicious input values and the payloads used to perform the attacks;

2. **Web Application Monitor**

   Server-side monitor that tracks the requests (attacks) performed to the target web application and the procedures made by the web application derived from these requests (e.g., queries to the databases, function calls). It has the following components:

   - **Target Web Application**: The web application in the testing scope;
   - **Attack Trace Collector**: Collects the traces of the requests made by the scanners and stores them in the trace database;
   - **Traces' DB**: The database used to store the collected traces. A trace is an execution path that begins at one (or more) entry points and ends at some point of the application which can be a sensitive sink;

3. **Attack Evaluator**

   Evaluates the successful attacks reported by the Ensemble Fuzzing module over the traces gathered by the Monitor, aiming thus to identify the source code of the vulnerabilities exploited and to associate with the exploit that allowed such. It has the following components:

   - **Attack Extractor**: Extracts the information of the attacks made by the scanners and associated with the vulnerabilities reported by those;
   - **Interesting Traces Identifier**: Using the attack information collected by the Attack Extractor, it is responsible for identifying the interesting traces (among all the stored traces) that are likely associated with a successful attack and that information. It inspects all the interesting traces collected and uses the web application's source code to perform a more in-depth analysis to confirm that an attack occurred in that trace by comparing the line of the trace with the correspondent source code line;
   - **Vulnerability Identifier**: Checks if the malicious payload from the attack reached any sensitive sink in its execution, evaluates the validity of the reported attacks and identifies the vulnerabilities in the source code;
   - **Final Report**: The report that aggregates the information found by the Vulnerability Identifier component, presenting if the attacks reported by the scanners exploited a vulnerability, the exploits used and the traces' fragments from the entry points until the sensitive sinks in which the attacks occurred. It highlights the entry points in which the attacks were first identified and the sensitive sinks they reached during the application's execution, including the source code file and lines executed in that case;

3.3 Main Modules

In this section, we provide a more in-depth description of the architecture's components.

3.3.1 Crawlers

The Crawlers are responsible for collecting the maximum number of requests possible (within the parameters defined by the user, e.g., the maximum call depth). They operate recursively through sending requests and getting the responses until exploring the whole attack surface of the target web application. They start by sending a request to the base URL of the web application, inspect the received response, search for hyperlinks and forms in this response and send new requests based on those, get their responses and continue this recursive process.

A request is an HTTP request without the header fields. It is composed of the method (e.g., GET, POST), the URL, which contains the path to a specific web application page and occasionally a query string with parameters and values associated, and, lastly, in some cases, some extra body data (also called request body data) which depends on the method. For example, for an HTTP POST method, the data is provided through this field. In contrast, in a GET method, the data is placed in the query string. The URL usually follows the following structure: <path>?<param>=<value>&<param>=<value>(...), where <path> represents the sequence of path segments that defines the location to a web page (e.g., http://www.example.com/index.php). The query string is optional but is usually used to send data by the GET method to query the application. It is preceded by a "?" and followed by parameter-value pairs delimited by "&", where <param> is the name of the parameter and <value> the data assigned to that parameter. The parameters are the names of the fields included in the HTML forms. Take as example the following request: /path/to/the/app?param1=value1¶m2=value2¶m3=value3, we easily associate it with the GET method and identify the query string and the data being sent. However, if we would be in the presence of a POST method, the body data from the request would be similar to the URL's query string, but it would not be sent directly in the URL and would be instead sent in the body field of the HTTP request.

The crawlers also have a built-in list with endpoints that usually appear in web applications (e.g., /phpMyAdmin, /.git), so they crawl through those. Furthermore, they usually use default values when filling the values of the URL's parameters or forms and must have a defined search depth to avoid unlimited crawling.

The requests are stored in the crawlers' request database of each crawler. These are of great importance because they will dictate whether a vulnerability is found or not when the scanners exercise the requests generated by crawlers. This means that if a crawler is not able to find out an endpoint that contains a vulnerability, it is guaranteed that the scanner will not be able to exploit such vulnerability it is not the responsibility of scanners to find new URLs to explore.

3.3.2 Request uniformizer

This module takes all requests from each crawler's database, converts them into a format that allows every scanner to use them, removes the duplicate requests and inserts the remaining unique requests into each database, respecting the database's specifications of each fuzzer's database (Uniformized Requests' DB in Figure 3.1) . Note that, as crawlers belong to an ensemble, different crawlers can produce similar or equal requests. Therefore, it is of the utmost importance to identify duplicated requests and remove them to get unique requests for better performance of the ensemble fuzzing.

The main challenge of this module is to get all the requests stored in crawlers' databases under different structures and specifications and uniformize these requests so that all the scanners can use them to perform attacks. Furthermore, the resultant uniformized requests must be inserted in each fuzzer's database. With this, we divided this module into three steps: **Extraction**, **Uniformization** and **Insertion**.

In the **Extraction** step, all the requests stored in each fuzzer's database are extracted, taking into account the database type and structure, as well as identified the valuable information that is worth retrieving, for example, the URL, the parameters and data sent in the request.

In the **Uniformization** step, the distinct requests are identified (i.e., the duplicate requests are eliminated) and then they are converted into a uniform format, which is the assembly of the information about the parameters and values collected in the Extraction step and that will allow different scanners to use them in the attack phase. The identification of the duplicate requests is made by comparing the values and parameters of the variables collected. Two requests are considered duplicate if they have exactly the same variables and values, and thus, are considered the same request. For instance, a GET request to `/example/param1=value1¶m2=value2` would only be considered duplicate if another GET request to `/example/param1=value1¶m2=value2` would happen to exist. In contrast, a POST request to `/example/param1=value1¶m2=value2` with the body data `bodyParam=value`, would be considered different from the previous GET request.

The uniformization process is the grouping of the query string and request body (if existant) parameters and values into the same format. Listing 3.1 presents an example of a uniformized request. It is a POST request to the page `/path/to/app`, with the parameter-value pairs `param1=h4cked`, `param2=pwned` and the parameter-value in the body `bodyParam=owned`. In order not to lose information about the requests, we consider the requests as a whole file and apply a base64 encoding to it, storing the result, which for this example would be: "UE9TVCAvcGF0aC90by9hcHA/cGFyYW0xPWg0Y2tlZCZwYXJhbTI9cHduZWQKCmJvZHlQYXJhbT1vd25lZA==".

Listing 3.1: Example of a uniformized request

```
POST /path/to/app?param1=h4cked&param2=pwned

bodyParam=owned
```

Finally, in the **Insertion** step, all the uniformized requests from the previous step are inserted in the database of each fuzzer, but first, all the crawler's initial requests and the requests from previous executions are deleted.

3.3.3 Scanners

Using the uniformized requests containing the URLs and the parameters-value pairs, the scanners recursively replace the values of the parameter-value pairs with malicious and malformed inputs, derived from the Fuzz Vectors and input mutation mechanisms they have, send the transformed request to the application, and search for the exploitation of vulnerabilities over the responses received from the application to these requests. The elements that the scanner looks for in the response depend on the type of vulnerability it wants to exploit. For instance, if the scanner is looking for reflected XSS vulnerabilities, an attack could look like the following: "`/path/to/the/app/?param=<script>alert('hacked')</script>`", with the malicious input being "`<script>alert('hacked')</script>`", known for possibly exploiting XSS vulnerabilities in vulnerable endpoints. In order to identify if the attack was successful, the scanner inspects the response from the web application and looks for evidence, usually searching for the attack in the response. In this case, the scanner checks if the malicious code is present in the response body and it was incorrectly encoded (or not encoded at all), and if it will be interpreted as JavaScript code by the browser; If so, it considers the attack successful.

When scanners find a successful attack, they add the vulnerability found to a report. Listing 3.2 presents an example of the information included in the report, where it is included the class of the vulnerability exploited, the method used in the request, the vulnerable URL, the variable used to exploit the vulnerability, the value used in the attack and the final exploit. Here, the exploit is the final request containing the malicious code, which was able to exploit a vulnerability.

Listing 3.2: Example of the reported information in the scanners' reports

```
Class: XSS
Method: GET
URL: http://example.com/path/vulnerable_to_xss/
Var: param
Attack: <script>alert("vulnerable")</script>
Exploit: http://example.com/path/vulnerable_to_xss/?param=<script>
   alert("vulnerable")</script>
```

3.3.4 Attack trace collector

This module collects the traces derived from the scanners' attacks and saves them in the Traces' DB for study (at later stages of the process).

To obtain the traces, the collector needs to monitor the web application and catch all web application instructions executed with the concrete values (i.e., the values sent in requests). As it is considered an extension of the web application's server, and, therefore, works like a debugger, it can have access to this information and collect it. Specifically, it has access to the payloads (i.e., the param-values) that arrive at the application, the changes in run-time, the function calls, variable assignments, return values, and the values sent back as a response to the scanner's requests.

Every time the application does a function call or a variable assignment, a call is also made to the attack trace collector module, allowing this module to analyse and collect the calls as soon as they are made. Despite having access to so much information, we only are interested in catching the function calls, variable assignments and occasionally the return values because that is the only useful information that we can use to identify the attacks in the traces. On the other hand, we exclude the values that would not bring any useful information (e.g., timestamps, memory usage), thus minimising the overload on our system. This overload happens because the trace generation is quantitatively heavy, and the number of lines generated for each trace can be huge. Therefore, in runtime, the collector identifies these kinds of instructions, removes them from the irrelevant information, and composes the traces with the resulting instructions. Listing 3.3 presents a source code example and the correspondent generated trace. The trace is delimited at the beginning and at the ending by the constants "`TRACE START`" and "`TRACE END`". After the starting delimiter in the trace, the following line states the main file that is executed (index.php). The following lines capture the execution of the source code line presented in the beginning of the listing, capturing the assignment of the variable *$a* and the call to the function *file_exists* with the value of *$a*. The details about the trace composition are given in Section 4.2.2.

Listing 3.3: Example of a source code and the correspondent trace

```
1. $a = 'example.php';
2. file_exists($a);
--------------------------------------------------------------------------
TRACE START [2021-06-28 12:58:12.729119]
1 2 {main}() /var/www/html/example/index.php:0
2 3 $a = `example.php' /var/www/html/example/index.php:1
2 2 file_exists($filename = 'example.php') /var/www/html/example/
   index.php:2
TRACE END   [2021-06-28 12:58:12.732321]
```

3.3.5 Attack extractor

This module is responsible for extracting the scanners' attacks associated with the vulnerabilities reported. As we already stated, the reports generated after the scanning phase contain information about the vulnerabilities exploited, the URLs in which the vulnerabilities were found and their parameters and values used in the attacks. The attack extraction is made by parsing these reports in order to extract the information related to the attacks and reconstruct them. However, in some specific cases (explained below), the attack extractor must also access to the Scanners' database to search for extra information not contained in the report, since sometimes the information provided in the reports is not enough to identify the attacks.

Usually, the attack extraction requires the reconstruction of the attack because the reports do not have direct information about the attack, i.e., all attack pieces are not together. Therefore, the extractor must recreate the attacks by parsing the reports to identify the pieces that compose an attack, namely the URL, parameters, and values sent to attain the attack with all the relevant information, extract them, and then put all together. For instance, the following: an attack that was made through the URL "http://vulnerable/example1", exploring the parameter "vulnerable_parameter" by sending the value "malicious" in a GET request. When the attack extractor extracts this kind of information, it constructs the attack as the following: "GET http://vulnerable/example1?vulnerable_parameter=malicious". While this attack reconstruction seems obvious, this information, in both GET and POST requests, is not directly available in some fuzzers.

For POST requests, the extractor accesses the Scanners' database in search of the complete information about the requests sent in the attack phase, looking for the missing pieces to recreate a complete attack. To do this, by taking into account the database type and columns used, it queries the database to retrieve the request body used in a specific attack. To find that specific attack, it uses the data contained in the report that is enough to pinpoint the attack (the URL, variable and payload used).

3.3.6 Interesting traces identifier

This module is responsible for identifying the traces that are considered interesting, i.e., that may be associated with an attack reconstructed by the attack extractor and reported by the Scanners.

This process is done by filtering the traces collected by the Monitor, one at a time, and comparing each trace's line with all the attacks collected by the attack extractor. Whether one attack is detected as part of the trace's line, the trace in the analysis is flagged as interesting and saved to be deeply inspected in the next phase. For example, supposing the module is analysing a trace's line of a given trace and finds the attack presented in the previous section, it will signalize that trace as an "interesting trace" and save it.

The process employed by this module works as an initial filter that quickly allows discarding the traces associated with no successful attacks and signalling interesting traces. Therefore, it will avoid

wasting resources in a more complex trace analysis that will get us no results in later phases.

However, this initial filtering can save traces where it was detected the same payload sent in a successful attack, despite the payload not exploiting vulnerabilities in these traces. This happens because when an attack is found in a trace line, the filter process stops, and the trace is considered interesting. Hence, this phase also goes a step further when identifying the attacks in the trace and applies a second filtering phase. As seen above, if an attack is successful in some endpoint, the same payload will be detected in all the traces where it appears, despite exploiting some vulnerability or not. This is because the trace line only has information about the number of the line of the code performed and the executed file. The vulnerability report only contains the attack performed and the URL to which the payload was sent. While, in most cases, the URL has similarities with the name of the correspondent file, we can not say for sure which file is being executed because the mapping and the URL redirection are application-specific. To handle this situation, another filter is performed to identify which are the interesting traces.

Take, for instance, the following URL: “ `http://example/randomPage`”, a scanner report with an entry with the following information: `{`**`URL:`**` "http://example/randomPage" | `**`Parameter:`**` "param1" | `**`Attack Value:`**` "malicious" }`, and two trace lines from different traces: `{$a = "malicious" /randomPage/file1.php:7}` and `{$b = "malicious" /randomPage/file2.php:5}`. With only this information, we can not say which trace line corresponds to the attack, and we can only identify the attack value in the traces present in both of the presented lines. However, these two lines correspond to two different files (file1.php, line 7, and file2.php, line5), and one may lead to the exploitation of a vulnerability and the other may not, leading to one real interesting trace and one false interesting trace. In this case, we do not know what leads to the execution of the two different files; It may be a parameter sent in the request that states which file to visit or even an internal variable that is set, but there are countless possibilities. To minimize this problem, the module employs a second filter, which we explain next.

The module, for each trace signalized as interesting by the first filter, it accesses the correspondent source code file and checks if that line corresponds to an entry point (e.g., $_POST. The current solution only considers as entry points the PHP superglobals). If so, it checks if the parameter in which the attack happened corresponds to the parameter found in the source code line; otherwise, it means that that attack did not happen in this trace, or at least it did not start in that line. Using the example above, if it checks the source code mentioned in the trace lines, it would have the additional information: `{file1.php:7 | $a = $_GET['param1'];}` and `{file2.php:5 | $b = $_GET['random'];}`. With this additional information, the module can distinguish the two trace lines and exclude the second one since the parameter from the entry point (`'random'`) differs from the parameter explored and reported by the scanner (`'param1'`).

Despite the second filter helping in effectively associating an attack with the correct trace, it may not able to distinguish all cases. For instance, if the same attack is sent in a different URL (that affects different files) and sent in a parameter with the same name, the filter will wrongly associate the attack to the trace. In the example above this would happen if the second trace line was `{file2.php:5 | $b`

= $_GET['param1'];}. This case will lead to inaccurate results since it will detect the attack in traces with no successful attacks and thus marking those traces as interesting. The evaluation phase will get incorrect results, in this specific case, since it will assess the validity of attacks in a trace that should not have been considered for analysis in the first place.

3.3.7 Vulnerability identifier

The Vulnerability identifier inspects the interesting traces identified in the previous phase with the attacks selected by the Attack Extractor to determine which attacks were successful, which traces are associated with them, and thus which vulnerabilities the traces contain.

The module starts by identifying the entry point of the attack, i.e., the first occurrence of the attack in the trace. From that point, it searches for the attack payload (i.e., the value used by the scanner to attack the application) until it reaches the end of the trace. If it identifies the payload in a sensitive sink, it considers the attack as successful. Oppositely, it can detect the attack in the trace, but it never reaches any sensitive sink existing in the trace. In that case, it considers the attack as unsuccessful. This means that if found an attack reported by scanners that did not exploit a vulnerability, i.e., a false positive from the scanner.

Furthermore, we currently only consider an attack successful if the payload enters the application from an entry point and reaches a function considered vulnerable to that attack type without being modified by the application in run-time. Since the module looks for the attack in the trace and does not track the changes (to that attack) that occur throughout the execution of the application, if the attack happens to be modified, the module will stop being able to identify it in the trace. Take for instance the XSS attack “<script>alert("malicious")</script>”. If we initially detect the attack and, during the application's execution, it is modified, for example, if it is HTML encoded (i.e., sanitized), the attack would become “<script>alert("hey")</script>”, and we would stop being able to detect it in the traces. If the attack only reached the sensitive sinks in the encoded format, we would not detect it, and the attack would be considered a false positive. We recall that we inspect the attacks reported by scanners as successful, and, hence, if an attack is not found in a trace, it will be considered as a false positive from scanners. Nevertheless, if, in our example, the attack would be eventually HTML decoded and converted back to the original format, we would be able to detect it once again.

Finally, to identify the vulnerabilities in the source code it uses the information contained in the trace line that indicates the file and line in the code of that specific trace. With this, by identifying (in the trace) a sensitive sink and the attack, it automatically has access to the source code line where the vulnerability is present.

3.4 Approach's Advantages

This section presents the advantages of our approach when identifying vulnerabilities in web applications.

We believe that our solution is versatile because, on the one hand, it takes advantage of the already studied and tuned web application fuzzers and shares their results with the other fuzzers without the need of tampering with any of their source code. With this, this solution allows for an unlimited number of web application fuzzers (for the chosen types of vulnerabilities), having only to adapt the output of that fuzzer to the same comparable format. On the other hand, it allows for the verification of the results discovered by the fuzzers using a tool to collect the web application traces without changing the web application's code.

Moreover, our ensemble fuzzing approach can achieve better overall results by improving the number of collected requests and sharing those with all the crawlers, resulting in more requests being executed by more scanners and an increase in the probability of exploitation. One can argue that a crawler with superior quality could achieve the same or better results than our Ensemble Fuzzing approach (in the Crawling phase). While this may be true, the time and effort spent in creating such a crawler would be far superior compared to simply adapting multiple open-source crawlers and sharing the results between them. We trust in the studies made by the creators of these fuzzers pass the responsibility of the correctness to them. We created our novel web application ensemble fuzzing approach, not thinking we would create an "ultimate fuzzer" but only an ensemble that could achieve better results than the standalone version of the fuzzers.

Our approach of confirming the fuzzer's results by analysing the traces generated by the attacks also has growth potential. By confirming the outputs of all the fuzzers, we may help to reduce some of the manual work required to inspect and confirm the results. Furthermore, by having access to the execution traces and the exact values that reached a function, our results can be more trustworthy than a manual inspection of the reports.

Finally, by not inspecting the source code of the web application (like static analysis approaches), our approach allows the use of the information obtained during the web application's execution, that contains data about the functions called and variables assigned, information which is not available in neither static nor dynamic analysis.

Chapter 4

Tool Design and Implementation

This chapter presents our current tool implementation of the proposed approach, which supports the evaluation of the developed work. It divides into the three modules that were described in the previous chapter. Section 4.1 describes the components and configurations used in our Ensemble Fuzzing module. Section 4.2 details the components used in the web application monitor, as well as the necessary modifications made to them in order to detect crucial function calls (e.g., sensitive sinks). Section 4.3 describes the processes used in the Attack Evaluator to validate the attacks in the execution traces and in the vulnerability detector to identify the code of the application of the vulnerabilities exploited. Lastly, Section 4.4 gives a complete usage example of the tool.

4.1 Ensemble Fuzzing

In our Ensemble Fuzzing solution, we focused on improving the final results of the fuzzers by sharing the results between them. As an initial step, we researched some of the non-commercial web application fuzzers in the wild and found six fuzzers that would be interesting to explore. In our fuzzer selection decision, we applied five criteria:

- Code availability;
- Crawling capabilities;
- SQLi and XSS attack capabilities;
- The ability to act as an authenticated user;
- Last update;

Table 4.1 contains a characterization of the assessed fuzzers for the five criteria.

Tool	Open-Source	Crawling	SQLi attacks	XSS attacks	Authentication	Latest Release
Burp Suite	✗	✓	✓	✓	✓	Nov 2020
OWASP ZAP	✓	✓	✓	✓	✓	Jan 2020
SkipFish	✓	✓	✓	✓	✓	Dec 2012
Wapiti3	✓	✓	✓	✓	✓	Feb 2020
XSStrike	✓	✓	✗	✓	✓	Dec 2019
w3af	✓	✓	✓	✓	✓	Apr 2015

Table 4.1: Characterization of the web application fuzzers evaluated to constitute the ensemble fuzzing.

All the fuzzers had the capability of crawling, running as authenticated users and performing XSS attacks. Despite the available Burp Suite community free version, the source code is not available to the users. XSStrike focused on XSS attacks and did not have the means to perform SQLi attacks, and the Skipfish fuzzer was too outdated. Based on this and in the Vimpari study [27], we ended up choosing the fuzzers *OWASP ZAP*, *Wapiti3* and *W3af*.

We implemented our ensemble approach with the three fuzzers, specifically the Wapiti 3.0.3, with the *Web_Spider* plugin for crawling, the w3af 2019.1.2, with the *Web_Spider* plugin for crawling, and the OWASP ZAP 2.9.0 (ZAP for short) with the *Spider* tool for crawling and the *active-scan* tool for attacking. For attacking, all tools were configured for *SQLi* and *XSS* attacks. ZAP was also configured with *xss_persistent* and *xss_reflected* modules. The plugin *xss* of the first two fuzzers also allows the discovery of persistent XSS.

One important aspect is that we did not change the source code of the crawlers and the scanners since our objective is not to improve the crawling and fuzzing capabilities of one specific fuzzer. We want to improve the overall results by sharing the information obtained by each fuzzer in the crawling phase that will lead to an improvement in the attacking phase.

4.1.1 Fuzzers' configuration

Many websites have some kind of authentication process that prevents unauthorized users from gaining access to sensitive information, and each of the explored fuzzers has its way of dealing with this issue. Also, the configurations of each fuzzer are application-specific, meaning they vary according to the target web applications.

ZAP has multiple ways of dealing with authentication, and we explored two of them (the ones that better suited our goals): *Form-Based* and *Script-Based*. In both, the user must supply some application context, like the username and password for the login and the logout pages (to avoid session loss). The *Form-based* mode automatically detects the data and format required for authentication, given the login

page. It also detects anti-CSRF tokens if the name of the tokens used by the application is in the ZAP Anti-CSRF list; if it is not, we must add it to the list. Although this automation seems useful, in some cases it is unable to find the correct data format, resulting in failed authentication attempts. Because of this, we opted for the *Script-Based* authentication mode for ZAP. In this mode, the user provides the application context and an authentication script, usually the POST data sent to the login page, that performs the actions required for authentication. This method achieves better results than the *Form-Based* mode; however, much more manual work is required, but it is something that is impossible to avoid.

Wapiti authentication is based on session cookies. The built-in utility *wapiti-getcookie*, given the login URL, can fetch the session cookies from the website that are later imported by the wapiti scanner. However, this utility does not detect the hidden values sent in the authentication process in some cases. To circumvent this issue, we created a script that generates the cookie with the mandatory values.

Regarding w3af, we initially explored two modes for the authentication configurations: *auto-complete* plugin and *cookie-based* authentication. Using the plugin, the user provides the login URL, authentication parameters, and information about a successful login. This plugin works most of the time; however, since there are applications where it does not, we opted for the *cookie-based* method, which is similar to the one from Wapiti. We manually get a cookie and convert it to a w3af-readable format.

Also, before crawling the web applications, a manual analysis is made to identify requests that can modify the application's state in an unwanted way. For example, the logout request or, as we used some applications created for security testing, usually some kind of endpoint allows the changing of the security level. Although this requires some manual work, it leads to more consistent results.

4.1.2 Crawlers

All the fuzzers in our selection differentiate between the crawler and the scanner, allowing us to run each one separately without changing the source code of the fuzzers.

The use of threads to perform the crawl phase influence the number of injection points and vulnerabilities found. For example, assuming a web page that is only accessible when some value is set, if the request that sets this value occurs after the request that tries to access that page, this attempt will be unsuccessful. In practice, if we execute the crawler in a multi-thread environment, we will get inconsistent results after each parallel execution. For comparison reasons and, consequently, for accurate results, we opted to apply single-threaded crawling when possible.

The ZAP fuzzer has an Application Programming Interface (API), listening to HTTP requests in a default port, allowing us to interact with ZAP programmatically. We created a script to automatize the creation of the authentication scripts mentioned previously and use the API to load these scripts and set the single thread environment. We also used *zap-cli* (the ZAP command-line interface) to interact with the fuzzer. In order to crawl the web application, we used the plugin *spider* with the context imported through the API calls. When the *spider* plugin completes its work, we use the session option to save the current session that contains all the information collected during the crawling phase.

The w3af fuzzer also has a command-line interface, *w3af_console*, that allows us to call it with a custom script (created manually, according to the specifications of the application) that sets the plugins and configurations to be used. The crawling script sets the plugin *web_spider* for the crawling phase, the URLs to be ignored by the crawler, the cookie created in the previous phase and sets the plugin *export_requests* to save all the requests in a CSV format.

Wapiti does not have a GUI, and we call the tool via command-line. We configured the tool using its flags, the cookie file generated previously, the URLs to ignore, set the scan with no modules (so it just runs the crawler) and stored the crawling session.

4.1.3 Request uniformizer

The Request Uniformizer is composed of three phases: **Extraction**, **Uniformization** and **Insertion**. Regarding the **Extraction** phase, for the ZAP fuzzer, we used the saved session from the crawling phase, which is stored in a HyperSQL database (HSQLDB), and the tool *SqlTool* to query the database in order to retrieve the requests. The results are saved in an HTML format, and we, posteriorly, convert them to a format where the requests are encoded in base64 not to miss any relevant information.

Similar to ZAP, Wapiti also has a saved session, a *SQLite* database containing the requests from the crawling phase. To access the contents of the database, we used the class *SqlPersister*[1], part of the Wapiti project, to get all the requests. Posteriorly we modified the format of those requests to match ZAP's encoding.

The requests from the w3af fuzzer were stored in a file by the *export_requests* plugin used in the Crawling phase, with each request being encoded in base64. By the end of this phase, we have all the requests from all the fuzzers in a base64 encoding.

For the **Uniformization** phase, the component leverages the internal functions of the fuzzers. For that, it uses the class *FuzzableRequest*[2] from the w3af internals. This allowed the representation of all the information collected in the Extraction phase into Python objects. In order to do so, we used the method *from_base64* to parse our base64 requests, collected in the previous phase, into *FuzzableRequest* objects.

We consider the requests uniformized because we were able to construct them as objects using the information collected in the Extraction phase and are now able to compare them equally. With this, we compare the requests with each other using the methods available in that class that allow us to compare the attributes of the objects. However, we had to make modifications in these functions' functionalities since they were not taking into account the order of the *POST* data parameters. For example, if we compared two requests with the same *POST* data, but with a different order in the parameters, the original functions would evaluate the requests differently. We did this change, firstly, because the parameters' order in

[1] https://github.com/wapiti-scanner/wapiti/blob/master/wapitiCore/net/sql_persister.py
[2] https://github.com/andresriancho/w3af/blob/master/w3af/core/data/request/fuzzable_request.py

the requests should not affect the outcome, i.e., a request to `/example?param1=v1¶m2=v2` and `/example?param2=v2¶m1=v1` should have the same output. Secondly, to obtain distinct requests and to decrease the total number of saved requests to the minimum possible, since the original functions would consider two requests with a different order in the POST parameters to be different and save both when in reality they were equivalent.

Having these functions, we iterate through the requests from each crawler and compare them with the already existent functions, saving the unique findings. At the end of this phase, we merge all the unique requests found and store them again in base64 format.

Finally, in the **Insertion** phase, we must insert the unique requests found back in each fuzzer's database. Regarding the *ZAP* fuzzer, we connect directly to the database, delete all the existing crawler requests and insert the uniformized requests, one by one, complying with the structure of the database. Since we used the *FuzzableRequest* class from w3af to represent our uniformized requests, we also use the methods from this class that allow us to get specific information about them.

For the *Wapiti* fuzzer, we use the *Web* class from the *Wapiti* internals, which contains a function that, given a raw HTTP request, converts it into a Wapiti specific format. We then use the *SqlitePersister* class to insert the requests into the saved database session. We overwrite the session that would be used in the attack phase with the current one, containing only the uniformized requests.

At last, for the *w3af* fuzzer, we did not have to access the database directly since the fuzzer has an option to set as input a base64 file containing the requests to be used in the attack phase. With this, we just had to pass as argument the file containing the uniformized requests, resultant from the Uniformization phase.

By the end of the three previous phases, all the fuzzers have the uniformized requests in their databases and are ready to start the *Scanning* phase.

4.1.4 Scanners

The scanners are responsible for taking the requests from their database and attacking the web application using them while using their built-in fuzz vectors (well-known values that can trigger specific vulnerabilities) and applying simple mutations in the data sent. This approach is based on the fuzzers used, and it is essential to note that if another set of fuzzers were to be used, the approach could differ. Also, we did not modify any of the fuzz vectors; each fuzzer uses its built-in data.

For the ZAP scanner, it was confirmed to load the uniformized database session using the *zap-cli* session load option. For the attack itself, it resorts of the *active-scan* plugin with the *sqli, xss, xss_reflected and xss_persistent* modules. To get the results, the *report* plugin to exports them as a XML report was used.

In the Wapiti scanner, the modules *permamentxss*, *sql* and *xss* are used for the attack and the flag –*skip-crawl* so that Wapiti does not execute the crawling phase. Lastly, the report from the attack is saved into a JSON file.

Finally, to import the results into the w3af scanner, the plugin *crawl import_results* was used with the base64 file generated in the previous phase, importing all the requests into the database, respecting the specifications. For the attack itself, the audit plugins *sqli* and *xss* were used. Finally, to generate the final report, the plugin *output xml_file* was used to generate an XML file with the attack details and vulnerabilities found.

By the end of this phase, we have the reports of the vulnerabilities found, one for each fuzzer. These reports contain the information of the attacks in which the scanner considered that some vulnerability was exploited.

4.2 Web Application Monitor

Regarding the Web Server, we used the Apache V.2.4.46 (Debian) HTTP server responsible for responding to the HTTP requests made to the target web applications.

The monitor itself is Xdebug, an extension from PHP with debugging capabilities that can collect information, in real-time, about the executions in the web application. The following section gives further details about the monitor, more specifically, the attack trace collector, and the changes we had to apply to Xdebug to serve our purposes.

4.2.1 Xdebug configuration

In order to be able to capture the execution traces, we used the mode *Trace* from Xdebug. This allowed the logging of function calls, the parameters and values used throughout the execution of the application and storing the execution traces. Specifically, we used the following configuration settings:

- *xdebug.mode=trace*: Used to activate the *Trace* function of Xdebug;
- *xdebug.start_with_request=yes*: To collect the traces from every request made to the application;
- *xdebug.collect_params=3*: To collect the variables and content passed in function calls;
- *xdebug.collect_assignments=1*: To collect all the variable assignments;
- *xdebug.trace_options=1*: To append to the output file instead of overwriting;
- *xdebug.var_display_max_data=-1*: To remove the limit in the maximum string length allowed to be appended to the output file;

These settings are written in the Apache configuration file and established when the HTTP server service is started.

4.2.2 Trace normalization

One initial change we had to do was related to the output format. In order to understand the problem, take the Listing 4.1 as the executed code and the Listing 4.2 as the output generated by Xdebug. This output contained information that we did not need, like the timestamp (in the first column) and the memory used (in the second column), so we removed those. It also identified function calls with the "→" symbol and the variable assignments with the "⇒" symbol. We converted those to a numeric notation to make identifying and making the data processing more straightforward. Finally, it identified the depth of the current function, i.e., the number of successive in-depth function calls. For example, if the function in the root file calls another function that consequently calls another function, the depth in the last function is three, in the middle function is two and in the root is one. The depth identification is based on the number of indented spaces outputted by Xdebug for call functions; so we also converted those to a numeric notation in order to be able to correctly identify the depth, since the string notation was ambiguous in some cases. Listing 4.3 presents the output of Listing 4.2 normalized with this changes applied. Looking at this output, a trace is delimited by the lines "TRACE START" and "TRACE END" delimiters, along with the timestamps associated, and contains between them a set of lines related to the executed code, being each one composed of four columns:

- The depth of the function;
- The instruction type: "1" for function return values, "2" for function calls, and "3" for variable assignments;
- The code of the executed instruction with the values in execution;
- The file of the executed instruction and the line of the code;

Listing 4.1: Example of a PHP snippet

```
1  <?php
2
3  function function_b () {
4      return "b".function_c();
5  }
6  function function_c() {
7      return "c";
8  }
9
10 $a = "string";
11 $b = function_b();
12
13 ?>
```

Listing 4.2: Trace generated by the execution of the code of Listing 4.1

```
1 TRACE START [2021-07-21 13:56:55.919700]
2     0.0005     365376   -> {main}() /var/www/html/DVWA/example.php:0
3                            => $a = 'string' /var/www/html/DVWA/example.php:10
4     0.0006     365376     -> function_b() /var/www/html/DVWA/example.php:11
5     0.0006     365376       -> function_c() /var/www/html/DVWA/example.php:4
6                            => $b = 'bc' /var/www/html/DVWA/example.php:11
7 TRACE END   [2021-07-21 13:56:55.919957]
```

Listing 4.3: Trace generated by the execution of the code of Listing 4.1 and normalized

```
1 TRACE START [2021-07-21 14:17:08.330332]
2 1 2 {main}() /var/www/html/DVWA/example.php:0
3 2 3 $a = 'string' /var/www/html/DVWA/example.php:10
4 2 2 function_b() /var/www/html/DVWA/example.php:11
5 3 2 function_c() /var/www/html/DVWA/example.php:4
6 2 3 $b = 'bc' /var/www/html/DVWA/example.php:11
7 TRACE END   [2021-07-21 14:17:08.330691]
```

4.2.3 Xdebug modification

With this format, we have already a good amount of information that can be used to detect successful attacks in the trace output. However, Xdebug is not able to trace calls to language constructs, i.e., instructions built into PHP itself, like *echo* and *exit*, that work differently from functions.

In our approach, we avoided modifying the source code of PHP internals. Instead, we developed a new version of Xdebug in order to achieve our goal of being able to log the *echo* and *exit* calls, essential for the detection of XSS vulnerabilities. With this in mind, we extended its functionalities by developing a hook that intercepts such instructions. For this, first, we had to analyse and understand the behaviour of Xdebug when executing the Web Application to detect the functions in the source code where we could code our hook. We observed that the Zend Virtual Machine (VM) calls the *statement_call handler*, a function exported by Xdebug mainly used for statistics matters (e.g., counting the number of lines executed) and debugging matters (mainly breakpoint checking). This function overwrites the Zend default handler and has some helpful information to work with, namely:

- The current file and line being executed;
- All the variables and respective values collected up until that point;
- The complete list of ZEND opcodes generated from that specific file;
- The stack frame, updated until the execution of the previous opcode executed, containing the variables and values up until that point;

Zend opcodes

Our main source of useful information is the list of opcodes provided by Zend. The Listing 4.4 displays an example of the opcodes generated by the execution of the code in Listing 4.1. We can see that, for each executed line, the first opcode is always "ZEND_EXT_STMT". Zend creates this opcode at compiling time and creates between statements, usually at the beginning of a new line. Also, this opcode triggers the call for the *statement_call* handler function, allowing Xdebug to stop at safe locations (e.g., after executing a line of code). In this example, two calls were made to the *statement_call handler* (lines 1 and 3). Regarding the remaining opcodes, the assignment instructions to variables *$a* and *$b* generate the opcode "ZEND_ASSIGN" (lines 2 and 6) and the function call to *function_b()* generates the opcodes "ZEND_INIT_FCALL" and "ZEND_DO_FCALL" (lines 4 and 5). Also, at the end of each function call there is always the "ZEND_RETURN" opcode (line 7).

Listing 4.4: Opcodes generated by the execution of the code of Listing 4.1.

```
1 Opcode: 101 (ZEND_EXT_STMT), File:Line: /var/www/html/DVWA/example.php:10
2 Opcode: 22 (ZEND_ASSIGN), File:Line: /var/www/html/DVWA/example.php:10
3 Opcode: 101 (ZEND_EXT_STMT), File:Line: /var/www/html/DVWA/example.php:11
4 Opcode: 61 (ZEND_INIT_FCALL), File:Line: /var/www/html/DVWA/example.php:11
5 Opcode: 60 (ZEND_DO_FCALL), File:Line: /var/www/html/DVWA/example.php:11
6 Opcode: 22 (ZEND_ASSIGN), File:Line: /var/www/html/DVWA/example.php:11
7 Opcode: 62 (ZEND_RETURN), File:Line: /var/www/html/DVWA/example.php:13
```

An opcode is a 4-tuple (opcode_type, op1, op2, result) representing the operation code type, the first operand, the second operand, and the result. The usage of these elements depends on the opcode type since not all opcodes use all the tuple elements (for instance, some opcodes only have one operand). However, for our purpose, we have interest in the opcodes "ZEND_ECHO" and "ZEND_EXIT", which represent the opcodes generated by the language constructs *echo* and *exit*, respectively, and are associated with XSS vulnerabilities, as we already stated.

ZEND_ECHO and ZEND_EXIT opcodes

An opcode contains two operands, however, the ZEND_ECHO and ZEND_EXIT opcodes only use one. This operand is an offset value that points to the pretended value and it can be one of four types:

1. *IS_CONST*: the operand has a **constant value** and the operand variable points to the constant value. Example: *echo* "Constant";

2. *IS_CV*: the operand is a **compiled variable** and the operand variable points to the slot in the stack frame where that variable is. Example: $var = "Var"; *echo* $var ;

3. *IS_TMP_VAR*: the operand is a **temporary variable** and the operand variable points to the slot in the stack frame where the variable is. Example: *echo* ($a = *addslashes*("Value")) **.** "Value" ;

4. *IS_VAR*: the operand is a **local variable** and the operand variable points to the slot in the stack frame where the variable is. Example: *echo addslashes*(“Value”);

Listing 4.5: Opcodes generated by the examples presented for each operand type

```
1  IS_CONST | echo "constant";
2  Opcode: 101 (ZEND_EXT_STMT), /var/www/html/DVWA/example.php:2
3  Opcode: 136 (ZEND_ECHO), /var/www/html/DVWA/example.php:2
4  Opcode: 62, (ZEND_RETURN) /var/www/html/DVWA/example.php:4
5
6  IS_CV | $var = "Var"; echo "var";
7  Opcode: 101 (ZEND_EXT_STMT), /var/www/html/DVWA/example.php:2
8  Opcode: 22 (ZEND_ASSIGN), /var/www/html/DVWA/example.php:2
9  Opcode: 101 (ZEND_EXT_STMT), /var/www/html/DVWA/example.php:3
10 Opcode: 136 (ZEND_ECHO), /var/www/html/DVWA/example.php:3
11 Opcode: 62, (ZEND_RETURN) /var/www/html/DVWA/example.php:5
12
13 IS_TMP_VAR | echo ($a = addslashes("Value")."Value");
14 Opcode: 101 (ZEND_EXT_STMT), /var/www/html/DVWA/example.php:2
15 Opcode: 61 (ZEND_INIT_FCALL), /var/www/html/DVWA/example.php:2
16 Opcode: 65 (ZEND_SEND_VAL), /var/www/html/DVWA/example.php:2
17 Opcode: 60 (ZEND_DO_FCALL), /var/www/html/DVWA/example.php:2
18 Opcode: 53 (ZEND_FAST_CONCAT), /var/www/html/DVWA/example.php:2
19 Opcode: 22 (ZEND_ASSIGN), /var/www/html/DVWA/example.php:2
20 Opcode: 136 (ZEND_ECHO), /var/www/html/DVWA/example.php:2
21 Opcode: 62, (ZEND_RETURN) /var/www/html/DVWA/example.php:4
22
23 IS_VAR | echo  addslashes("Value");
24 Opcode: 101 (ZEND_EXT_STMT) /var/www/html/DVWA/example.php:2
25 Opcode: 61 (ZEND_INIT_FCALL), /var/www/html/DVWA/example.php:2
26 Opcode: 65 (ZEND_SEND_VAL), /var/www/html/DVWA/example.php:2
27 Opcode: 60 (ZEND_DO_FCALL), /var/www/html/DVWA/example.php:2
28 Opcode: 136 (ZEND_ECHO) /var/www/html/DVWA/example.php:2
29 Opcode: 62, (ZEND_RETURN) /var/www/html/DVWA/example.php:4
```

Getting the values of the constant and compiled variables (cases 1 and 2) was not difficult. The former value (IS_CONST) is stored in adjacent memory to the opcode tuple in the Zend compiling phase; So, accessing to it is direct. The latter (IS_CV) points to a location in the stack memory of the variable whose value was stored with the execution of the previous opcodes, and, therefore, it is in need to find out where is the “IS_CV” operand.

Listing 4.5 presents the generated opcodes for each type presented above. Regarding case 1 (lines 1-4), the opcode 136 (ZEND_ECHO), in line 3, has as its first operand the value to be printed (in this case the string “constant”) and so it can get the value easily. For case 2 (lines 6-11), the ZEND_ECHO opcode (line 10) has as its first operand the value that is set in the ZEND’s previous execution (lines 7-8), where there is made the assign (line 8) of the value “Var” to the variable “$var”. The operand of the ZEND_ECHO opcode points to the value of the set variable, so it can also get the value.

The challenge here is obtaining the values of the temporary and local variables (cases 3 and 4) only with the information until the execution of the previous opcode, which is the information we have

available at that point in the execution. Since we observe the opcode list and stack frame when a "ZEND_EXT_STMT" is found, if there are function calls happening in the same line in opcodes following this one, we are not able to get the result values directly because that would require information about the execution of all the opcodes executed in the line being inspected. To address this challenge, we validate if the ZEND VM function will be called an additional time. If so, we skip the execution and get the value in the next call to this function, ensuring that the value complies with the executed code in the previous execution. If we can not determine if the function will be called again, we follow a bottom-up approach and inspect the former opcodes to get the values used.

For instance, in Listing 4.5, for case 3 (lines 13-21), there is only one ZEND_EXT_STMT opcode, meaning there will only be made one function call to the Xdebug function. When it tries to get the value of what was printed in the ZEND_ECHO opcode (line 20), the operand points to the result of the variable assignment made in the previous lines (line 19). However, because it is made in the same line as the ZEND_ECHO opcode, the operand is still pointing to a rubbish value in the execute data stack since it is not updated with the execution data of the current line. Because of this, it is forced to backtrack the origin of the value, iterating through the opcodes of line number 2 (/var/www/html/DVWA/example.php:2) in search for the value. With this, it inspects the operand of the ZEND_ASSIGN opcode, which assigns one value to a variable. So, one of the operands points to the value to be assigned, which again is the result of the concatenation (opcode ZEND_FAST_CONCAT in line 17) of two values, one of which is a function call (opcodes ZEND_INIT_FCALL, ZEND_SEND_VAL, ZEND_DO_FCALL in lines 15-17) and the other is a constant which is present in one of its operands.

Case 4 is very similar to case 3 since we also have an opcode ZEND_ECHO (line 28), which has an operand that points to the result of the function call (to the function *addslashes("Value")*), represented in lines 25-27. Hence, with a similar approach as in case 3, we can address case 4. This shows how the algorithm we propose has to iterate through the opcodes in search for the value used in the ZEND_ECHO opcodes, to be able to capture the *echo* function calls and their value. As we have to deal with different opcodes, we have to consider the possibility of jump opcodes since those are associated with conditionals and hence define which opcodes get executed and which ones are skipped. Listing 4.6 illustrates an example of the opcodes generated by the question mark assign, the ternary operator. Taking the PHP code in lines 1 and 2, the opcodes generated by Zend for line 2 are the ones in lines 7-12. In the question mark assign, there is a boolean value evaluated (in variable $bool), and if it is evaluated to *True*, it is executed the code following the "?", otherwise it is executed the code following the ":". The opcodes reflect this behaviour; in line 8, a jump opcode has as its first operand the value to evaluate, and in its second operand the offset to jump to (the ZEND_QM_ASSIGN of line 11); in line 10, an unconditional jump opcode that always jumps to the offset (the ZEND_ECHO of line 12). We cannot directly get the value used by the ZEND_ECHO opcode in line 12 because it is from a function executed in the same line of code we are currently checking. As we already stated, we have access to the concrete values determined in previous lines of code. As a first approach to handle this issue, we could check the offset from where it is trying to obtain the value and look for that offset in the results from the previous opcodes,

but it happens that the offset value would be the same in the ZEND_QM_ASSIGN opcodes, from lines 9 and 11. Therefore, to get the one used at execution time, we must check the jump opcodes and determine the jumps made to know which opcodes were executed.

Listing 4.6: Opcodes generated by the PHP question mark.

```
$bool = True;
echo $bool ? "A" : "B";
--------------------------
Opcodes generated:
Opcode: 101 (ZEND_EXT_STMT)
Opcode: 22 (ZEND_ASSIGN)
Opcode: 101 (ZEND_EXT_STMT)
Opcode: 43 (ZEND_JMPZ)
Opcode: 31 (ZEND_QM_ASSIGN)
Opcode: 42 (ZEND_JMP)
Opcode: 31 (ZEND_QM_ASSIGN)
Opcode: 136 (ZEND_ECHO)
Opcode: 62 (ZEND_RETURN)
```

Algorithm to trace Echo and Exit

The approach of tracing the language constructs *Echo* and *Exit* is reflected in Algorithm 1. Here, most of the initial information comes from the current *execution_data* structure passed from Zend to Xdebug. The algorithm gets the current file line being executed and the opcode list from the entire file being executed. Then, it iterates the array from the current index saved in a global variable until it reaches an opcode that is no longer in the current line. It observes the type of the opcode and searches for the opcodes responsible for the *Echo* and *Exit* functions. Only the first operand is used for these functions, and it corresponds to the string being printed, so it checks the type of this first operand. If it is from a type that it can immediately get the value, it does it (lines 10-11). If it is not, it identifies the opcodes executed after the Xdebug call and determines if there will be another call to Xdebug, by observing the opcodes following the ones being executed at the moment (line 18). If it determines that there will be another call, it saves the current opcode index and gets the value in the next call to Xdebug (lines 18-20).

In the next iteration, it checks if it has skipped the previous execution, and if so, it can safely get the value pointed by the operand variables, knowing that it will contain the correct value, which is in line with the execution data from the previous iteration (lines 13-16). In case of the function not being called another time, the algorithm tries to get the values immediately by inspecting the opcodes that precede the current one, as the final value will be derived from those executions (lines 21-22). In this matter, the Zend VM has an extensive file that handles all the opcodes and the types of operands and combinations between them. However, these functions that handled the opcode execution could not be called because they would eventually call the modified statement_call handler in Xdebug and originate a call loop. To circumvent this situation, we adapted some of the Zend VM opcode handlers that are usually used with

the *echo* and *exit* language constructs and can get the values for those opcodes. Our implementation currently has handlers for 14 opcodes: “ZEND_ASSIGN” ,“ZEND_CONCAT”, “ZEND_DEFINED”, “ZEND_FAST_CONCAT”, “ZEND_FETCH_CONSTANT”, “ZEND_FETCH_DIM_R”, “ZEND_FETCH_IS”, “ZEND_FETCH_R”, “ZEND_ISSET_ISEMPTY_CV”, “ZEND_IS_EQUAL”, “ZEND_IS_IDENTICAL”, “ZEND_IS_NOT_EQUAL”, “ZEND_QM_ASSIGN” and “ZEND_ROPE_END”.

The algorithm must also take into account possible jumps. So, if it stumbles upon any jump opcode it checks if its jump is based on a boolean evaluation (line 26). If it is an unconditional jump, it can immediately jump to the correspondent opcode (line 36). If it has an evaluation, it checks if it is doable to get that value, and if it is, jumps to it (lines 27-28). If it is not able to get the value directly, it checks for the existence of ZEND_EXIT opcodes since an *exit* function may be reached, and no further opcodes are processed (line 30). If it finds it, traces a custom message that possibly a *exit* function was reached and activates a flag that indicates to the ZEND Engine to trace the next function return value (lines 31-34). With this, in a further phase, where we will study the generated traces, we will be able to identify a possible *exit* and confirm if it did indeed happen by checking the adjacent generated traces and comparing the traced result value with the attack.

In both cases, it can get a printable value with which it will be able to write in the trace file. The function **create_trace_entry(printable_value)** (lines 23,33) creates the structure used by Xdebug to trace the user-defined and internal functions and fills it with the required information, along with the printable value created. At last, the function **write_in_trace_file(function_entry)** (lines 24,34) calls the Xdebug function that appends the entry to the trace file.

With this process, the Xdebug is now able to trace the function calls made to the web application with the addition of the code constructs *echo* and *exit*, and we can start inspecting the generated traces.

Algorithm 1: Making Xdebug trace the language constructs *Echo* and *Exit*

```
1  Xdebug_Statement_Handler(execution_data, g_current_index, g_skipped)
2      current_line ← execution_data.current_line;
3      opcode_list ← execution_data.opcode_list;
4      current_index ← g_current_index;
5      while opcode_list[current_index].line = current_line do
6          opcode ← opcode_list[current_index];
7          switch opcode.opcode_type
8              case ZEND_ECHO || ZEND_EXIT
9                  opcode_oper1 ← opcode.operand1 ;
10                 if opcode_oper1.operand_type is "IS_CONST" or "IS_CV" then
11                     printable_value ← opcode_oper1.value
12                 else if opcode_oper1.operand_type is "IS_TMP_VAR" or "IS_VAR" then
13                     if g_skipped then
14                         printable_value ← opcode_oper1.value ;
15                         current_index ← update_index_to_current_line(current_line) ;
16                         g_skipped ← false;
17                     else
18                         if can_skip(opcode_list, current_index) then
19                             g_current_index ← current_index;
20                             return;
21                         else
22                             printable_value ← get_value_from_opcodes_current_line(current_line,
                                 opcode_list, current_index);
23                     function_entry ← create_trace_entry(printable_value);
24                     write_in_trace_file(function_entry);
25             case Jump opcode
26                 if is_conditional_jump(opcode) then
27                     if get_value_conditional_jump(opcode) then
28                         current_index ← get_index_from_jump(opcode);
29                     else
30                         if line_contains_zend_exit(opcode_list) then
31                             printable_value ← possible_zend_exit();
32                             trace_next_function_return();
33                             function_entry ← create_trace_entry(printable_value);
34                             write_in_trace_file(function_entry);
35                 else
36                     current_index ← get_index_from_jump(opcode);
37         ++current_index;
38     end
39 End
```

4.3 Attack Evaluator

The Attack Evaluator module is responsible for identifying if the attacks made by the Scanners that they classified as successful are capable of exploiting some vulnerability and identifies the code of the application associated with these vulnerabilities. For that, the module correlates the conclusions detailed in the vulnerability reports with the traces collected by the Monitor, and then identifies over the resulting traces the vulnerable code. It is divided into three sub-modules: **Attack Extractor**, **Interesting Traces Identifier** and **Vulnerability Identifier**.

4.3.1 Attack extractor

This module is responsible for extracting the requests used in the Scanning Phase that successfully exploited vulnerabilities. This is made by parsing the vulnerability reports and extracting relevant information regarding the requests.

The implementation of the attack extractor resorts on some Python libraries to achieve this, namely the *xml* library for the ZAP and w3af reports, and the *json* library for the Wapiti report. The module performs these actions as follows. Firstly, it only collects the information of the selected vulnerabilities, namely XSS and SQLi. Also, for all fuzzers, it has to ensure all the relevant information, namely the method (e.g., GET, POST), the URL, the attack payload, the parameter variable used in the attack and the exploit were properly obtained. Listings 4.7, 4.8 and 4.9 present a simplified example of an output snippet of the w3af, ZAP and Wapiti reports, respectively.

Listing 4.7: Example of w3af scanner's report

```
1  <vulnerability id="[138]" method="GET" name="SQL injection" plugin="sqli" severity="High" url
       ="http://127.0.0.1/DVWA/vulnerabilities/brute/" var="username">
2  <http-request>
3      <status>GET http://127.0.0.1/DVWA/vulnerabilities/brute/?Login=Login&amp;help_button=View
           %20Help&amp;password=&amp;source_button=View%20Source&amp;username=1%272%223 HTTP
           /1.1</status>
4      <headers>
5          <header field="Content-length" content="0" />
6          <header field="Accept-encoding" content="gzip, deflate" />
7          <header field="Accept" content="*/*" />
8          <header field="User-agent" content="w3af.org" />
9          <header field="Host" content="127.0.0.1" />
10         <header field="Referer" content="http://127.0.0.1/" />
11         <header field="Cookie" content="security=low; PHPSESSID=o0cjfp4a2gr4ure6eklftav2v0" />
12         <header field="Content-type" content="application/x-www-form-urlencoded" />
13     </headers>
14     <body content-encoding="base64"></body>
15 </http-request>
16 </vulnerability>
```

Listing 4.8: Example of ZAP scanner's report

```
1  <alert>Cross Site Scripting (Reflected)</alert>
2  <instances>
3  <instance>
4  <uri>http://127.0.0.1/DVWA/vulnerabilities/xss_s/</uri>
5  <method>POST</method>
6  <param>mtxMessage</param>
7  <attack>&lt;/div&gt;&lt;script&gt;alert(1);&lt;/script&gt;&lt;div&gt;</attack>
8  <evidence>&lt;/div&gt;&lt;script&gt;alert(1);&lt;/script&gt;&lt;div&gt;</evidence>
9  </instance>
10 </instances>
```

Listing 4.9: Example of Wapiti scanner's report

```
1    "vulnerabilities": {
2      "SQL Injection": [
3        {
4          "method": "GET",
5          "path": "/DVWA/vulnerabilities/brute/",
6          "info": "MySQL Injection via injection in the parameter username",
7          "level": 1,
8          "parameter": "username",
9          "http_request": "GET /DVWA/vulnerabilities/brute/?Login=Login&help_button=View+Help&
               password=&source_button=View+Source&username=%C2%BF%27%22%28 HTTP/1.1\nHost:
               127.0.0.1",
10         "curl_command": "curl \"http://127.0.0.1/DVWA/vulnerabilities/brute/?Login=Login&
               help_button=View+Help&password=&source_button=View+Source&username=%C2%BF
               %27%22%28\""
11       }
12     ]
13   }
```

Regarding the w3af's report, the "vulnerability" tag (line 1) has the majority of the relevant information, namely the type of request, the type of vulnerability, the parameter used and the URL of the attack. The "status" tag (line 3) has more details about the request made to the web application, especially useful is GET requests. Finally, the "body" tag (line 14) contains the body content information, useful in the POST requests.

For the ZAP's report, the "alert" tag (line 1) specifies the vulnerability type, the "uri" tag (line 4) contains the URL to which the attack was sent, the "method" tag (line 5) the type of the request made, the "param" tag (line 6) the parameter used in the attack and the "attack" tag (line 7) contains the attack sent. However, the ZAP's reports do not always contain the exploit, meaning that in some cases, it is not possible to "construct" the exploit by only inspecting the ZAP's report. For instance, exploits resulting from GET requests where the exploit is usually the URL itself the report contains the whole exploit. But for cases like POST requests, we do not have the requests' body information, and thus we can not extract the exploit used by the ZAP Scanner using only the reports. To face this, we must access the ZAP's database to find the exploit since the ZAP fuzzer identifies the requests that successfully exploited

a vulnerability in its database. To do so, we use the URL, variable and attack information from the report to extract the exploit from ZAP's database.

Lastly, contrary to the other two, Wapiti's report has the output in a JSON format, and the vulnerabilities are grouped in-depth. The "vulnerabilities" tag (line 1) has a list of vulnerability types; In this example, we only have the SQLi type (line 2). The "method" tag has the type of the request, the "path" tag contains the path in the URL to which the request was sent, the "parameter" tag has the parameter used in the attack and the "http_request" tag contains the HTTP request sent in the request. Usually the "parameter" and "http_request" tags are enough to extract all the relevant information.

To obtain all the relevant information in the reports, we created three different parsers, one for each scanner. As seen, each report has its own format of data representation, but the content is similar in all the scanners, making it easy to create an attack list with similar content. By the end of this phase, we have an attack list for each fuzzer, where each element contains all the relevant information mentioned above.

4.3.2 Interesting traces identifier

Before starting the attack search in the trace database, we perform an analysis to identify the interesting traces (the ones that can contain an attack) to reduce the number of total traces that fully impact the next module (see next section).

The Algorithm 2 presents the followed approach. It starts by grouping all the trace files and iterating through each one (line 5). Since each trace file contains multiple traces, it identifies the trace delimiter for the beginning (line 6) and ending (line 15) of a trace, using regular expressions to achieve that purpose.

If it is the beginning of a new trace, it immediately checks the first URL visited by the Scanner in that trace, present in the subsequent line (line 7), and checks if an attack occurred in that URL, saving the attacks that did (lines 9-11). If no attacks were found to that specific URL, it skips that trace (lines 13-14).

While it does not reach the end of the trace, it iterates through the created *current_attack_list* generated at the beginning of the trace and checks if the attack is possibly present in the current line (lines 18-19). To do this, it checks if the attack is a sub-string of the line, and if it is, it calls the function "compareTraceLineWithSourceCode(line)" (line 20) in order to further confirm that the attack happened in that trace. If positive, the attack is saved together with the trace (line 21). The attack is removed from the current attack list because it was already found in that trace, and no further search is required (line 22). When it reaches the end of the trace, by identifying the ending tag, it just skips to the next trace and continues the process.

Algorithm 2: Identifying Interesting Traces

```
    Input  : trace_database,trace_starting_tag, trace_ending_tag, attack_list;
    Output : final_list;
 1  final_list ← [];
 2  foreach trace in trace_database do
 3      current_index ← 0;
 4      current_attack_list ← [];
 5      for line in trace[current_index] do
 6          if line is trace_starting_tag then
 7              line ← trace[++current_index];
 8              root_url ← line.getURL();
 9              foreach attack in attack_list do
10                  if attack in root_url then
11                      current_attack_list.append(attack);
12              end
13              if current_attack_list.isEmpty() then
14                  break;
15          else if line is trace_ending_tag then
16              break;
17          else
18              for attack in current attack list do
19                  if attack is present in line then
20                      if compareTraceLineWithSourceCode(line) then
21                          final_list.append(trace,attack);
22                          current_attack_list.remove(attack);
23              end
24          ++current_index;
25      end
26  end
```

If it made a simple substring search in each line, it would risk misidentifying an attack in a trace. This can happen because the changes made by fuzzers in each consecutive attack are minimal, meaning that many attacks can be substrings of other attacks, and so it may identify the shorter attacks in traces where the attack was actually lengthier. To prevent this, the function **compareTraceLineWithSource-Code(line)** compares the current trace line with the source code line that was executed in the trace. We took this additional step because the tested fuzzers usually have similar attacks, differing only by a few characters. This sometimes led to erroneous identification of the attacks that could be present in some traces. Take for instance the example in Listing 4.10. In here, we have Attacks 1 and 2 as the attacks made

in the Scanning Phase, and the *vulnerable_function*, a function vulnerable to the Attack1. If we were to identify if an Attack was present in the function call only by analysing the trace line, we would identify the *Attack2* as the attack that exploited this *vulnerable_function*, which is not the case. By comparing with the source code line, we can determine the exact values used in that function call. In this case, we can see that the prime character was added to the vulnerable variable and conclude that Attack1 was used in this trace line.

Listing 4.10: Necessity of analysing the source code of the respective trace line

```
1 Attack1: h4cked
2 Attack2: 'h4cked
3
4 Trace line: vulnerable_function($arg1 = 'h4cked)
5 Source code line: vulnerable_function("'" . $vulnerable_var)
```

Thus, to solve this problem, the algorithm of the function CompareTraceLineWithSourceCode compares the trace line with the source code line, returns the list of variables used in the source code line and the values assigned in the trace. Afterwards, it checks if the attack is present in any of those values. To achieve this, we followed the Algorithm 3 that initially writes the trace line into a PHP file in order to take advantage of the PHP internal functions that allow it to make a better comparison. Namely, it take advantage of the internal tokenization functions to convert both the trace line and the source code line into tokens to compare them later.

Algorithm 3: Comparing the trace line with the source code

1. Write the trace line into a PHP file;
2. Tokenize the trace file and extract the list of tokens from the expected line;
3. Tokenize the source code file and extract the list of tokens from the expected line;
4. Iterate through both tokens' lists and identify the value used in each variable;

It uses the function *file_get_contents* to read a PHP file into a String and the function *token_get_all* used by the Zend engine's lexical scanner to tokenize the code in the string into PHP tokens. Two lists of tokens are presented in Listings 4.11 and 4.12 and the former has the list of tokens of a traced function call while the latter has the correspondent source code.

Listing 4.11: Tokens generated by the PHP code: test_function($arg1 = ’localhost’);

```
1 [0] => T_STRING:test_function
2 [1] => UNKNOWN:(
3 [2] => T_VARIABLE:$arg1
4 [3] => UNKNOWN:=
5 [4] => T_CONSTANT_ENCAPSED_STRING:'localhost'
6 [5] => UNKNOWN:)
7 [6] => UNKNOWN:;
```

Listing 4.12: Tokens generated by the PHP code: test_function($server);

```
[0] => T_STRING:test_function
[1] => UNKNOWN:(
[2] => T_VARIABLE:$server
[3] => UNKNOWN:)
[4] => UNKNOWN:;
```

It initially iterates through the source code token list and identifies the index in which each function begins and ends and the variables used, since a function can have recursive function calls as parameters. After this, it iteratively compares both lists and infers the values of the parameters used. For the Listings 4.11 and 4.12 the output of the list comparison function would be “{$server, ‘localhost’}”, meaning that the variable $server was assigned the value ‘localhost’ in the trace. Finally, it would compare all the values assigned to the variables and compare them to the attack to check if the Scanners could inject the attack and capture it in the trace.

There are, however, limitations to this method since we may not have enough information to infer the values of the variables. Take for instance the code of Listing 4.13. Here we have in line 1 the source code and in line 2 the value in the execution trace. If we were to infer the values used in the variables $b and $c, we would not be able to do so without any extra information, leading to an ambiguous situation. Here we have a combination of five possible solutions ({$b = ‘’,$c= ‘text’}, {$b = ‘t’,$c= ‘ext’}, {$b = ‘te’,$c= ‘xt’}, {$b = ‘tex’,$c= ‘t’}, {$b = ‘text’,$c= ‘’}). Having only the source code line and the trace line makes it impossible to know which is the right solution since this would require knowledge about the execution data until that point, which we do not have access to at this stage. To partially solve this problem, we assume that all the variables can assume the whole value. In this case, our custom solution would be {$b = ‘text’,$c= ‘text’} and we flag this result so that later when we look for the attack in this kind of custom solution, we only check if the attack is a sub-string of any variable in the solution. This has the disadvantage of increasing the final number of false positives. It leads to the original problem that leads to the consequent increase in the false detection of attacks in the traces.

While it may be possible to do it without inspecting the application’s source code, we need to do

it to improve the precision in identifying the reported successful attacks in the traces. The source code inspection is related to the similarities in the payloads (i.e., the injected inputs) used by the fuzzers that requires the inspection of the source code lines of the traces' lines in which it is suspected that an attack is present. By not inspecting the source code, we increase the number of wrongly identified attacks and a final increase in false positives.

Listing 4.13: Impossibility of infering the values used in each variable in the source code

```
1 $a = $b.$c;
2 $a = 'text';
```

By the end of this phase, we have a list of attacks in which it is identified the fuzzer that did the attack, the parameters in which the attack was sent and the trace in which this attack possibly happened. Each element in the list has the following information:

- **Fuzzer:** The fuzzer that exploited the vulnerability;
- **Vulnerability type:** The type of vulnerability exploited (e.g., SQLi);
- **Attack URL:** The URL where the vulnerability was exploited;
- **Parameter:** The parameter where the attack payload was sent;
- **Attack:** The attack payload;
- **Trace:** The trace in which the attack was identified;

4.3.3 Vulnerability identifier

In this phase, we have the attack list resultant from the previous phase. Having this information, this phase is responsible for confirming if the attacks did indeed reach a sensitive sink and exploited some vulnerability. Also, it reports the lines of the source code for the vulnerabilities found.

Our approach is presented through Algorithm 4. It firstly iterates through the attack list from the previous section (line 2) and iterates through each line of the trace in each element of that list (line 8).

Algorithm 4: Identifying the Vulnerability Exploitation in the Traces

```
   Input   : attack_list;
   Output  : vulnerability_list;
1  vulnerability_list ← [];
2  foreach element in attack_list do
3      entry_point ← ∅ ;
4      current_index ← 0;
5      found_TP ← false;
6      attack ← element.getAttack();
7      trace ← element.getTrace();
8      for line in trace[current_index] do
9          if attack in line then
10             line_type ← line.getType();
11             if line_type is FUNCTION_CALL then
12                 if entry_point is ∅ then
13                     entry_point ← line;
14                 if entry point is not ∅ then
15                     if checkForVulnerability(attack,line) then
16                         vulnerability_list.add("TP",element);
17                         found_TP ← true;
18             else if line_type is VARIABLE_ASSIGNMENT then
19                 if entry_point is ∅ then
20                     entry_point ← line;
21             else if line_type is RETURN_VALUE then
22                 if checkForVulnerability(attack, trace[current_index-1]) then
23                     vulnerability_list.add("TP",element);
24                     found_TP ← true;
25         ++current_index;
26     end
27     if found_TP is false then
28         vulnerability_list.add("FP",element);
29 end
```

As seen previously, the trace line can be one of three types: function call, variable assignment and return value. The approach differs based on the type of the current line, namely:

- **Function Call** (lines 11-17): If the entry point has still not been identified, it checks if the current function call acted as the entry point for the attack. If the entry point has already been found, it

checks if the attack reached a function vulnerable to it, and if so, it considers this attack to be successful;

- **Variable Assignment** (lines 18-20): In this case, it only checks if this is a possible entry point for the attack. This is because, in the selected vulnerability types, the attacks happen in specific vulnerable function calls. If the value assigned to a variable results from a function call, the trace would have another line that represented that function call. So it does not need to check for vulnerabilities in the trace line that represent variable assignments;

- **Return value** (lines 21-24): The function call "trace_next_function_return" presented in Algorithm 1, line 32, allows for the occurrence of trace lines containing the return of a function. Since this only happens if a *exit* function could be reached, it means the previous trace line may contain a vulnerable function (function calls and return values are always adjacent in trace files) in which the attack was injected. So, the algorithm calls the function to check for vulnerabilities that checks if the attack present in the previous line reached a sensitive sink.

The **checkForVulnerability(attack, line)** function checks if the attack is possibly exploiting some vulnerability in the current line. Since we have the attacks made by the Scanner and the type of vulnerabilities that they exploited, obtained in the Attack Extractor phase (Section 4.3.1), we use this information to infer if the attack actually exploited a vulnerability in this line. This depends on the type of vulnerability, and for each type our approach differs, namely:

- **SQLi:** It considered that the attack is successful if it can find it in one of the following known SQLi vulnerable functions: "mysql_query", "mysql_unbufered_query", "mysql_db_query", "mysqli_query", "mysqli_real_query", "mysqli_master_query", "mysqli_multi_query", "mysqli_stmt_execute" and "mysqli_execute".

- **XSS:** Similarly, for the XSS vulnerabilities we considered the attack successful if the reached one of the following functions known to be XSS vulnerable: "echo", "print", "printf", "die", "error", "exit", "file_put_contents", "file_get_contents" and "var_dump". Aditionally, to distinguish between the reflected and stored XSS vulnerabilities, we added the following constraints:

  1. A Reflected XSS attack is only considered if the entry point of the attack is one of the PHP Superglobals[3](e.g., $_GET, $_POST). If it is not, we consider that it did not happen from the user interaction and then it is not considered an attack.
  2. Contrarily, for the Stored XSS attacks, we only consider them if the value of the attack does **not** come from a PHP Superglobal.

[3] https://www.php.net/manual/en/language.variables.superglobals.php

These constraints allowed us to differentiate between the Reflected and Stored XSS attacks since the former requires the attack to successfully reach a sensitive sink in the same execution trace in which the attack is inserted in the attacked system. In contrast, the latter requires the attack to exist in the system, meaning that it cannot be resultant from any function requiring user interaction in the current execution trace.

4.4 Usage Example

Consider a web application which is composed of two files: **file1.php** and **file2.php**. The main page of the application is the file1, and the file2 cannot be requested directly. Listings 4.14 and 4.15 present the source code of file1.php and file2.php, respectively. The application will be tested using 3 fuzzers: **FuzzerA**, **FuzzerB** and **FuzzerC**.

Listing 4.14: Source code of the example file **file1.php**

```
1 include 'file2.php';
2 $a = $_GET['variable'];
3 echo $a;
4 if($_GET['continue'] == 'yes'){
5     vulnerable_function_in_file2($a);
6 }
```

Listing 4.15: Source code of the example file **file2.php**

```
1 function vulnerable_function_in_file2($param) {
2     exit($param);
3 }
```

An example of execution of our current solution follows the subsequent order:

1. The user manually identifies the parameters and values used in the authentication mechanism, such as usernames, passwords, and CSRF tokens. Furthermore, endpoints that could cause sessions to be terminated should also be identified to achieve successful scans in the authenticated pages. This does not apply to our simple example since there is no authentication;

2. Each fuzzer runs its crawler in the web application, collecting the pages, searching for new endpoints and recursively exploring the web application. In our example, consider that the crawler from fuzzer1 is scanner1, the crawler from fuzzer2 is crawler2, and the crawler from fuzzer3 is crawler3. For instance, consider that crawler1 found the URL `"/file1.php?variable=crawler1&continue=yes"`, crawler2 found the URL

"/file1.php?variable=crawler2" and fuzzer3 found the URL "/file1.php". Each crawler stored the found request in its fuzzer's database;

3. The Request Uniformizer module takes all the requests collected by the crawlers and proceeds to uniformize them. In this example, all the requests are different and, since the requests are originated from GET requests, there is no body data to compare. Having this, no request will be excluded in this phase. By the end of this phase, all the requests are inserted in each fuzzers' database and all the scanners now have access to the requests collected;

4. Each scanner runs on the web application using the requests given by the Request Uniformizer module. To keep the simplicity of the example, and since the subsequent phases of the process run sequentially, we continue the example with only the execution of one scanner. Consider that the scanner from fuzzer1 is scanner1, and for the sake of the example, consider that it reported 2 XSS vulnerabilities, detailed in Listing 4.16;

Listing 4.16: Example of report generated by scanner1

```
1  <alert>Cross Site Scripting (Reflected)</alert>
2  <instances>
3  <instance>
4  <uri>/file1.php?variable=<script>alert(1)</script></uri>
5  <method>GET</method>
6  <param>variable</param>
7  <attack>&lt;script&gt;alert(1);&lt;/script&gt;</attack>
8  <evidence>&gt;&lt;script&gt;alert(1);&lt;/script&gt;</evidence>
9  </instance>
10 <instance>
11 <uri>/file1.php?variable=<script>alert(2)</script>&continue=yes</uri>
12 <method>GET</method>
13 <param>variable</param>
14 <attack>&lt;script&gt;alert(2);&lt;/script&gt;</attack>
15 <evidence>&gt;&lt;script&gt;alert(2);&lt;/script&gt;</evidence>
16 </instance>
17 </instances>
```

5. During the attacks' execution, the monitor monitors and logs all the requests made to the web application. Consider that, for the execution of scanner1, the traces generated are the ones presented in Listing 4.17. All four traces correspond to the attacks the scanner1 made to the web application;

Listing 4.17: Traces generated in the attack phase of scanner1

```
TRACE START [2021-09-21 00:00:01.000000]
1 2 {main}() /file1.php:0
2 3 $a = 'value' /file1.php:2
2 2 echo($var = 'value') /file1.php:3
TRACE END   [2021-09-21 00:00:02.000000]

TRACE START [2021-09-21 00:00:02.000000]
1 2 {main}() /file1.php:0
2 3 $a = 'anothervalue' /file1.php:2
2 2 echo($var = 'anothervalue') /file1.php:3
TRACE END   [2021-09-21 00:00:03.000000]

TRACE START [2021-09-21 00:00:03.000000]
1 2 {main}() /file1.php:0
2 3 $a = '<script>alert(1)</script>' /file1.php:2
2 2 echo($var = '<script>alert(1)</script>') /file1.php:3
TRACE END   [2021-09-21 00:00:04.000000]

TRACE START [2021-09-21 00:00:04.000000]
1 2 {main}() /file1.php:0
2 3 $a = '<script>alert(2)</script>' /file1.php:2
2 2 echo($var = '<script>alert(2)</script>') /file1.php:3
3 2 vulnerable_function_in_file2($var = '<script>alert(2)</script>') /file1.php:5
3 3 exit($var = '<script>alert(2)</script>') /file2.php:2
TRACE END   [2021-09-21 00:00:05.000000]
```

6. Subsequently, the Attack Extractor phase extracts all the attacks made by scanner1 from the respective report. There is no need to access the scanners' database since the attacks are from GET requests and do not require the additional information present in the database since the reports already contain all the information required to "build" the complete attack;

7. In the Interesting Traces Identifier phase, the traces present in Listing 4.17 are compared with the attacks reported, present in Listing 4.16. Here we can see that there are two distinct attacks, with two different payloads, "`<script>alert(1)</script>`" exploited in the variable "`variable`", and a second attack with the payload "`<script>alert(2)</script>`", exploited in the variable "`variable`". The first two traces will be excluded because they do not contain any of the two payloads above. Regarding the last two traces, when any of the payloads is found, the tool will mark that occurrence as the entry point of the attack and check the source code line of the trace line where the exploit is found. In our example, lines 16, 22 and 24 will be compared with the source code to confirm that that specific trace is from the reported attack. With this, line 16 and 22 will be compared with "`echo $a;`" and line 24 with "`exit($param)`". Since there is only one variable being printed in both `echo` and `exit` functions (and so, there is no concatenation of values), the tool considers the attack payload found in the trace to correspond to the one reported, and so the last two traces will be considered interesting;

8. In the next phase, the Vulnerability Identifier takes the (two) interesting traces identified in the previous phase and identifies if the attack payload reached sensitive sinks. In this case, since the attack reported is from the XSS type, it will look for functions like `echo` and `exit`. Our example will detect that the attack payload reached a sensitive sink in trace lines 16, 22, and 24;

9. Finally, it generates a report with the results found. Our example will confirm the two vulnerabilities reported by the scanner and will report a third one that was not reported. This third reported vulnerability is the result of two vulnerabilities exploited with one single scanner attack. The attack present in Listing 4.16, lines 10-16, is reported as exploiting one vulnerability because the scanner can detect the sent payload in the responses from the server without sanitization. However, with the trace information, our solution can detect that, with the same payload, two sensitive sinks are reached, one in line 22 and one in line 24 in Listing 4.17. The output of the tool regarding this example and scanner is presented in Listing 4.18. Here, the tool presents the results as true positives, displays the fuzzer of which the attack was made, the attack sent, the variable exploited, the entry point in the source code file where the attack was first seen, the sensitive sink reached and the file and line in which it is present, the exploit that allows the replay of the attack in the web application, encoded in base64, and the trace in which the vulnerability was detected.

Listing 4.18: Output of the tool regarding the attacks of scanner1

```
1  TP | Fuzzer:fuzzer1 | Type: XSS_R | Attack: <script>alert(1)</script> | Var: ``variable"
       | Entry_Point: ``/file1.php:2" | Reached_function: echo in file1.php:3 | Exploit: ``
       R0VUIC9maWxlMS5waHA/dmFyaWFibGU9PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0PiZjb250aW51ZT15ZXM
       =" | Trace:
2      1 2 {main}() /file1.php:0
3      2 3 $a = '<script>alert(1)</script>' /file1.php:2
4      2 2 echo($var = '<script>alert(1)</script>') /file1.php:3
5  TP | Fuzzer:fuzzer1 | Type: XSS_R | Attack: <script>alert(2)</script> | Var: ``variable"
       | Entry_Point: ``/file1.php:2" | Reached_function: exit in file2.php:2 | Exploit: ``
       R0VUIC9maWxlMS5waHA/dmFyaWFibGU9PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0PiZjb250aW51ZT15ZXM
       =" | Trace:
6      1 2 {main}() /file1.php:0
7      2 3 $a = '<script>alert(2)</script>' /file1.php:2
8      2 2 echo($var = '<script>alert(2)</script>') /file1.php:3
9      3 2 vulnerable_function_in_file2($var = '<script>alert(2)</script>') /file1.php:5
10     3 3 exit($var = '<script>alert(2)</script>') /file2.php:2
11 TP | Fuzzer:fuzzer1 | Type: XSS_R | Attack: <script>alert(2)</script> | Var: ``variable"
       | Entry_Point: ``/file1.php:2" | Reached_function: echo in file1.php:3 | Exploit: ``
       R0VUIC9maWxlMS5waHA/dmFyaWFibGU9PHNjcmlwdD5hbGVydCgyKTwvc2NyaXB0Pg==" | Trace:
12     1 2 {main}() /file1.php:0
13     2 3 $a = '<script>alert(2)</script>' /file1.php:2
14     2 2 echo($var = '<script>alert(2)</script>') /file1.php:3
15     3 2 vulnerable_function_in_file2($var = '<script>alert(2)</script>') /file1.php:5
16     3 3 exit($var = '<script>alert(2)</script>') /file2.php:2
```

Chapter 5

Evaluation

We performed a set of experiments to evaluate the efficiency of our designed and implemented approach in detecting XSS and SQLi vulnerabilities and confirming the truthfulness of the results by studying the generated execution traces.

The goal of the evaluation was to answer the following questions and to verify if our approach resolved the challenges presented in Section 3.1:

(1) Can the ensemble fuzzing lead to the discovery of different endpoints in the tested web applications that other crawlers would miss in their standalone version?

(2) Can the ensemble fuzzing lead to the discovery of vulnerabilities that would be missed if the fuzzers used only the requests found by their crawler?

(3) Can the ensemble fuzzing improve the overall coverage and precision of the vulnerability detection?

(4) Is the solution able to identify vulnerabilities from the attacks reported by fuzzers and their traces?

(5) Has the solution the ability to signalize false positives over the attacks reported by fuzzers?

With this, we divided the evaluation into two phases: the Ensemble Fuzzing Evaluation, described in Section 5.2, and the Trace Analysis Evaluation and vulnerability identification, described in Section 5.3. Moreover, Section 5.2 intends to answer the first three questions, associated with the two first challenges: Fuzzers' exploitation capabilities and Code coverage (see Sections 3.1.1 and 3.1.2. Section 5.3 aims to answer the remaining evaluation questions, which are associated with the challenges of extraction of traces and identification of vulnerabilities (see Sections 3.1.3 and 3.1.4).

Section 5.1 gives an overview of the web applications used during the experiments.

5.1 Web Applications

This section provides some context about the selected web applications. We used three open-source web applications with multiple vulnerabilities of our chosen classes (XSS and SQLi) to test our solution.

5.1.1 DVWA

The first tested web application is Damn Vulnerable Web Application (DVWA)[1], a PHP/MySQL web application designed to be vulnerable to SQL injections, XSS and other vulnerability classes and used mainly for educational purposes by students and security professionals.

DVWA was chosen as a starting point because it is a simple application that has the vulnerabilities very well documented and allows for a very fast analysis of the source code. Despite the extensive range of present vulnerabilities, we focused on the XSS and SQLi vulnerabilities.

The application requires user authentication, as very few pages can be accessed without an active session, and the different pages can be accessed via simple hyperlink tags inside of list items. It also has the option to change the level of security used, which, for our testing purposes, was set to the level "low" (no security).

5.1.2 Mutillidae

Our second tested app is OWASP Mutillidae[2], which is also a deliberately vulnerable web application that contains at least one vulnerability type belonging to OWASP Top 10 (from the 2017 version) [48]. It has been used in graduate security courses, corporate web security courses and as a target for vulnerability assessment software.

In this app, access to its different pages is made using nested HTML unordered lists that change with *onmouseover* events. There is a base URL, and most of the pages are accessible by changing the value of a variable reserved indicating the web page in the query string.

5.1.3 bWAPP

Lastly, we tested Buggy Web Application (bWAPP)[3], a vulnerable PHP web application with over one hundred vulnerabilities, including those described in OWASP Top 10. Like the applications above, it has well-documented vulnerabilities and is used by security professionals to test their skills and tools. Like DVWA and Mutillidae, it also requires user authentication and the pages that do not require authentication are few and vulnerability free.

[1] https://www.dvwa.co.uk/
[2] https://github.com/webpwnized/mutillidae
[3] https://www.itsecgames.com/

5.2 Ensemble Fuzzing Evaluation

The next two sections present and discuss the crawlers and scanners' evaluation into the ensemble and individually.

5.2.1 Crawler evaluation

The main objective in this phase is to evaluate the crawler's from the different fuzzers and understand the main differences in the URLs collected and the discrepancies found.

The evaluation of the crawlers was based on the results provided by the Request Uniformizer module, where we compared the requests executed by each crawler, pursuing the following comparison criteria:

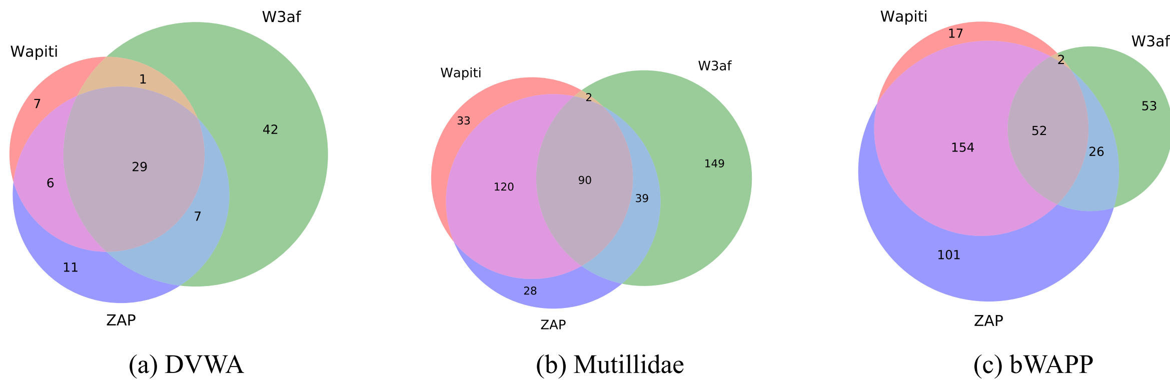
1. Requests with the same method (e.g., GET);
2. Requests with the same base URL;
3. Requests with the same variables and the same values in the query string, despite order;
4. Requests with the same POST data;

These criteria are used to identify identical requests. This means that if two requests pass all the criteria, they are considered identical, and, therefore, we consider them to be the same request.

Regarding the third criterion, we initially considered that requests that had the same variables in the query string, despite its value, were considered similar. Although most of the requests with the same variables usually lead to the same page, in some cases, the value determined the presented page. To avoid these missing endpoints, we compared the values of these variables. For criterion 4, two requests are only considered equal if they have exactly the same POST data.

Based on these four criteria, we automatically compared the requests made by all crawlers and verified which ones represented the unique and similar requests between each crawler. Figure 5.1 presents the results of our comparison by web application. Here, each crawler is represented by a circle. The numbers represent the number of requests found, and the numbers within multiple circles (the intersections) represent the requests found by all the fuzzers of those circles.

As we can observe, in general, no crawler stands out because they have different results depending on the tested web application. For instance, the rate of common requests discovered by all crawlers was 28% (29) in DVWA, 20% (90) in Mutillidae, and 13% (52) in bWAPP. However, the number of equal requests outputted by two crawlers was more significant than the previous rates. For example, for Multilidae and bWAPP, it was, respectively, 35% (161 out of 461) and 45% (182 out of 405). The more interesting cases are the requests found only by one crawler, which we called *crawler's unique requests*. Almost or even more than 50% of the requests found by each crawler were only discovered by it. In DVWA, *crawler's unique requests* correspond to 58% (60 out of 103) of the requests, in Mutillidae 46% (210 out of 461)



(a) DVWA (b) Mutillidae (c) bWAPP

Figure 5.1: Number of URLs found by crawlers of each fuzzer when inspecting each web application.

and bWAPP 42% (171 out of 405). Note that this considerable number of *crawler's unique requests* is due to the criteria used. There are very similar requests that are considered different from the requests from other crawlers because they used different values in the parameters of the requests. For instance, there are cases where the crawlers used values directly related to it (e.g., the crawler from w3af sending the value "w3af" in the parameters).

Regarding the *crawler's unique requests* of DVWA, ZAP's requests are related to the login page and pages accessible from that endpoint, requests which w3af and Wapiti excluded from their results. Of the few *crawler's unique requests* found by Wapiti, one has an entry point only accessible through a Javascript function redirect. ZAP misses this endpoint completely, and w3af reaches the function but excludes the URL due to an erroneous redirect. w3af unique requests are mainly images that are missed or ignored by the other fuzzers.

For the Multilidae and bWAPP applications, the unique ZAP and Wapiti requests are related to a public *phpmyadmin* service that they were able to detect. In contrast, the w3af's ones are related to folders that have common names, such as "/javascript" or "/webservices" that the other fuzzers do not explore by default, as well as some URLs found by the other fuzzers, but with different values and parameters.

These crawling results already denote the existing discrepancies in what crawlers can discover. Different crawlers achieve different results depending on the tested application. For instance, in the DVWA and Mutillidae, the w3af fuzzer seems to achieve a more significant number of unique requests. However, in Mutillidae, the number of requests shared only between the ZAP and Wapiti fuzzers is also considerable. On the other hand, in the bWAPP application, the w3af seems to be the worst fuzzer among all three, making ZAP the chosen best in this case. This proves the advantages of our approach since no crawler stands out in all three web applications. If that was the case, maybe the ensemble fuzzing would not achieve such significant results because a single crawler would achieve that without sharing the collected information with all the other crawlers.

Given the results and analysis, we can state that there are advantages in crawling applications with an ensemble fuzzing because there are requests missed by some crawlers (that can contain vulnerabilities)

and that would not be found by the fuzzers whose crawlers missed the requests. Therefore, given the results we can answer positively to question (1).

As a final and important consideration we pose about the unique requests, notice that they are the ones that will make the difference in the final results of our approach because these are the requests that will allow the scanners whose crawlers have not found the request to explore these requests and possibly exploit some vulnerability. Therefore, these are the requests which we will focus more of our attention from now on.

5.2.2 Scanner evaluation

In this section, it is assessed the capabilities of the scanners individually and in the ensemble version. Also, the section intends to identify if the scanners can explore requests found by other crawlers than their own and, hence, exploit vulnerabilities from them.

For the individual evaluation, each scanner runs as standalone with the requests that its crawler discovered. In the ensemble version, each scanner runs with the requests found by the ensemble. Hence, having a list of requests gotten by the three fuzzers' crawlers, each fuzzer's scanner is fed with the list and uses the requests by exercising them with diverse inputs and carries out the attacks on the web applications to try the exploitation of SQLi and XSS vulnerabilities.

Table 5.1 summarizes the results obtained from the scanners on both evaluations, where columns 3, 5 and 7 regard the individual fuzzers and columns 4, 6, and 8 regarding the ensemble fuzzing (EF) version. The last column represents the false positives that were manually identified.

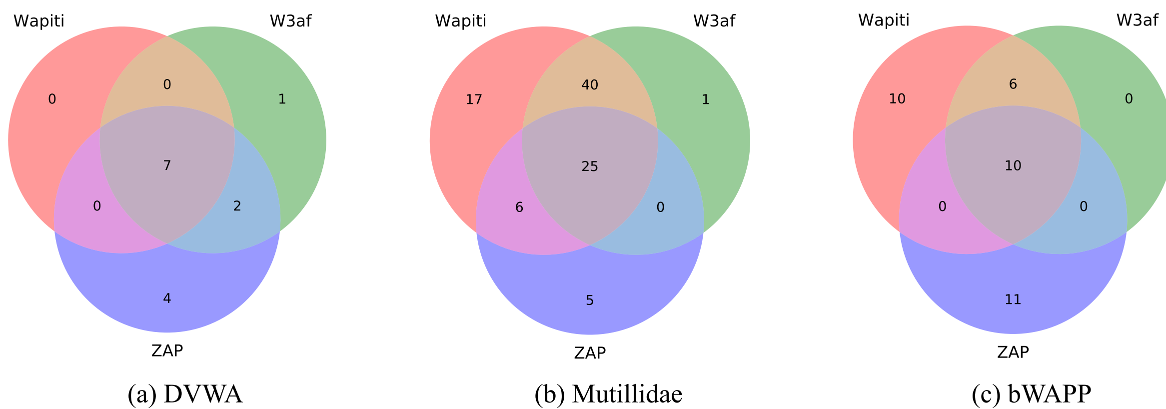
Table 5.1: Number of successfully exploited vulnerabilities in the tested web applications.

Web App	**Vulnerability**	**Wapiti**	**Wapiti-EF**	**w3af**	**w3af-EF**	**ZAP**	**ZAP-EF**	**False Positives**
DVWA	SQLi	2	2	3	3	3	3	1
	Reflected XSS	5	5	1	5	7	8	1
	Stored XSS	0	0	0	2	2	2	0
Mutillidae	SQLi	17	17	4	12	9	9	0
	Reflected XSS	52	54	26	43	22	24	1
	Stored XSS	17	17	1	11	3	3	1
bWAPP	SQLi	1	1	0	1	0	0	0
	Reflected XSS	9	21	1	13	19	20	11
	Stored XSS	1	4	0	2	1	1	0

We can see that the results vary with the complexity of each web application because no scanner is the best in all the web applications. It is visible that scanners within the ensemble have improved their precision in discovering vulnerabilities, denoting thus that they can explore requests found by other crawlers.

W3af-EF is the fuzzer that increased its precision the most since it has the most significant increase in the number of successfully exploited vulnerabilities, comparing the standalone version with the ensemble. Wapiti, on the other hand, has a less significant increase, although not null. ZAP-EF exploited more vulnerabilities in DVWA, while Wapiti-EF has better results in the Mutillidae and bWAPP applications.

In order to assess the quality of the scanners, we manually compared the reported vulnerabilities between each other, identifying the unique and common findings between all the scanners. Figure 5.2 displays the distribution of the vulnerability findings for each application. The number of common findings represents 50% (7 out of 14) in DVWA, 27% (25 out of 94) in Mutillidae and 27% (10 out of 37) in bWAPP. Furthermore, Wapiti-EF and ZAP-EF had the most significant number of unique findings, varying according to the application tested. w3af-EF, on the other hand, had a low rate of unique findings (1 or 0).



(a) DVWA (b) Mutillidae (c) bWAPP

Figure 5.2: Vulnerability distribution by each fuzzer when attacking each web application.

During the analysis of the reported vulnerabilities, we found interesting cases when we compared the results of the scanners with the requests made by the crawlers. For this specific analysis, we defined three types of interesting cases:

1. The crawler of the fuzzer F_x missed the URL_i, and the remaining crawlers found it but could not exploit any vulnerability in it. In the ensemble version, only F_x was able to explore the URL_i and exploit the vulnerability;

2. The crawler of the fuzzer F_x was the only one able to find the URL_i and exploit vulnerabilities in it. In the ensemble version, all the fuzzers were able to exploit vulnerabilities in URL_i;

3. Both fuzzers F_x and F_y found the same base URL; however, F_y could not retrieve a valid request from it because it missed a variable or sent an erroneous value. On the other hand, F_x discovered the correct request (URL_i) and exploited vulnerabilities in it. In the ensemble, F_y was also able to exploit vulnerabilities in URL_i.

The first case is the one that is of most interest to our approach. This is the case where the ensemble version can catch vulnerabilities that would be missed whether each scanner was to run in its standalone versions. This is because the only scanner that can exploit the vulnerability is the one whose crawler could not find the URL. The second case, while not as interesting as the first, denotes the cases where the scanners could not exploit the vulnerabilities, not because they did not have the capabilities to do so, but because the crawlers were not able to catch that URL. Finally, the last case is interesting to denote the importance of sending all the required parameters in the requests and how that affects the final results of the fuzzer.

We found 18 interesting cases, namely 6 from the first case, 11 from the second and 1 from the last. As an example for each case, Table 5.2 presents an interesting XSS vulnerability for each one. We used a binary representation to identify if the vulnerability was explored or not, and the ”–EF” columns represent the attacks performed in the ensemble. In contrast, the other columns represent the attacks performed by the scanner individually. Also, the second row represents an example for the first interesting case presented above, and so on.

Table 5.2: An example of XSS of each case.

URL	Wapiti	Wapiti-EF	w3af	w3af-EF	ZAP	ZAP-EF
/bWAPP/csrf_3.php	0	1	0	0	0	0
/DVWA/vulnerabilities/ csrf/test_credentials.php	1	1	0	1	0	1
/bWAPP/xss_php_self.php	0	1	0	1	1	1

The first row represents a vulnerability where the request was missed by the Wapiti’s crawler but was caught by another crawler that could not exploit that vulnerability. In the ensemble, only the scanner from Wapiti could exploit this vulnerability that would otherwise be missed in the web application assessment. By inspecting the application code, a submit button sends some values, including one in a hidden input, which is updated according to the response from the server, and only Wapiti was able to manage this hidden input, unlike the other fuzzers.

In the second row, Wapiti’s crawler is the only one that founds the URL and can exploit vulnerabilities in it. When this URL is shared between the remaining fuzzers, all of them can exploit the vulnerability.

In the last row, the w3af’s crawler missed the URL, but the remaining crawlers could detect it. However, ZAP and Wapiti returned different parameters for this URL. Wapiti only found the parameters from a form used to update the information, missing a crucial parameter required for the request to be accepted by the application; hence, its exploitation was unsuccessful. Oh the other hand, ZAP found the correct URL and explored it successfully. When the correct URL (from ZAP) was given to all the other scanners, they all could explore the vulnerability. In this example, Wapiti sent a request to “`/bWAPP/xss_php_self.php?firstname=default&lastname=default`”, while ZAP sent

to "/bWAPP/xss_php_self.php?firstname=ZAP&lastname=ZAP**&form=submit**". The code that validates this specific request is presented in Listing 5.1. Here, the code vulnerable to XSS (line 7) can only be accessible if all the parameters *form*, *firstname* and *lastname* are set, making the parameter *form*, discovered by ZAP, crucial in the vulnerability exploitation.

Listing 5.1: Source code that validates the requests sent to /bWAPP/xss_php_self.php

```
1 if(isset($_GET["form"]) && isset($_GET["firstname"]) && isset($_GET["lastname"])) {
2   $firstname = $_GET["firstname"];
3   $lastname = $_GET["lastname"];
4   if($firstname == "" or $lastname == "")
5     echo "<font color=\"red\">Please enter both fields...</font>";
6   else
7         echo "Welcome ".xss($firstname)." ".xss($lastname);
8 }
```

During the analysis, the attacks resultant from the scanner's reports were manually performed and analysed in the web applications to assess their truthfulness. We found 15 false positives, as depicted in the last collumn of Table 5.1, namely: 13 from ZAP and 2 from w3af. Most of the ZAP false positives were XSS attacks in the application *bWAPP*, where the values used in the variables instantiated by ZAP were not used since it was only checked if the variable was set. The fuzzer wrongly identified the attacks in the responses received from the web application, confused them with other attacks made to other pages of the same web application, and wrongly associated it with the attacks made to that page. One example of a w3af false positive was in DVWA, as it reported an SQLi attack in "/DVWA/instructions.php?**doc=**", where the attacked parameter was "doc". Listing 5.2 presents a fragment of the code that deals with this parameter. Here, the value of the variable is only used to compare with the values of an existing array (lines 1-4), and if it does not match any of its values (line 6), a default one is used (line 7). In this case, as doc='' (empty string), the $selectedDocId variable takes the "readme" default value. Later, in line 9, $readFile receives the name of the file (README.md) based on the value of $selectedDocId. Since the value obtained from the entry point is only used to get the value from the array, no SQLi attack is possible.

Based on the obtained results and observations we made, we can state that question (2) has a positive response.

Listing 5.2: Source code of the page attacked by the w3af fuzzer that resulted in a false positive.

```
1 $docs = array(
2         'readme'=> array('file' => 'README.md'),
3         'PDF'=> array('file' => 'docs/pdf.html'),
4 );
5 $selectedDocId = isset($_GET['doc']) ? $_GET['doc'] : '';
6 if( !array_key_exists($selectedDocId, $docs) ) {
7         $selectedDocId = 'readme';
8 }
9 $readFile = $docs[$selectedDocId]['file'];
```

5.3 Trace Analysis Evaluation

In this section is evaluated the capabilities of the proposed trace analysis solution in correctly identifying the successful exploitation of vulnerabilities with the scanners' attacks, as well as the false positives detected in the manual analysis to the scanners' results and the vulnerabilities in the source code. Before we proceed with the evaluation, we defined a set of criteria to measure the results.

5.3.1 Metrics

To evaluate the effectiveness of the trace analysis solution, we needed to correlate the solution's output with the scanners' output in order to calculate the final output. Such correlation must happen because the solution's output does not have knowledge about the manual analysis made to the scanners. Also, we needed to make some considerations about the results obtained by scanners and the solution. Take for instance an example where the solution states that an attack does not exploit any vulnerability and so it is considered a false positive (FP) arised from scanners. We recall that the scanner considers that any attack it reports is a true positive (TP), but after a manual inspection, the attack could be revealed as not exploiting any vulnerability, and so being a false positive of the scanner. However, the final output of the solution would be a true negative (TN) because it confirmed that an attack was indeed a scanner's false positive. On the other hand, an effective true positive reported by scanners might not be correctly checked by the solution, originating thus a false negative (FN) for this latter. Therefore, we split the scanner results as TP and FP (after the manual analysis) and the evaluation of the results of the solution can be TP, FN, TN or FP. Hence, we evaluated the final results of our solution based on the confusion matrix presented in Table 5.3 and the following assumptions:

Table 5.3: Confusion matrix for the final results

		Solution Output	
		TP	FP
Scanners' Output	TP	**TP**	**FN**
	FP	**FP**	**TN**

- **True Positive (TP)**: If both the fuzzer and our solution identify the vulnerability as a true positive, the final result is also a true positive, because both the fuzzer and our solution are in an agreement;
- **False Negative (FN)**: If our solution is not able to confirm the exploitation of the vulnerability, and there is, in fact a vulnerability, the final result is a false negative for the solution;
- **False Positive (FP)**: If our solution identifies as being a successful exploitation of a vulnerability, and the fuzzer wrongly identified a vulnerability, the final result is a false positive for the solution;

- **True Negative (TN)**: If our solution is not able to detect as a successful exploitation of a vulnerability and the fuzzer actually wrongly identified a vulnerability, the final result is a true negative for the solution and it was able to confirm a false positive of the fuzzers.

Furthermore, to measure the effectiveness of our solution, we used the following metrics:

- **Accuracy** = $\frac{TP+TN}{TP+FP+TN+FN}$: Used to measure the ratio of correct predictions over the total number of instances evaluated;
- **Precision** = $\frac{TP}{TP+FP}$: Used to measure the positive patterns that are correctly predicted from the total predicted patterns in a positive class;
- **Recall** = $\frac{TP}{TP+FN}$: Used to measure the fraction of positive patterns that are correctly classified;
- **F-Score** = $\frac{2*Recall*Precision}{Recall+Precision}$: Used to present the harmonic mean between the recall and precision values;

5.3.2 Evaluation

Initially, in our evaluation, we started by comparing the data generated with the execution traces. Table 5.4 presents the number of files and traces generated, and the total size, in the attack phase of each fuzzer into the ensemble fuzzing, for all the tested web applications. Here we can see that the number of generated files does not directly reflect the number of represented traces. For instance, in the DVWA application, both Wapiti and ZAP generated 8 files. However, Wapiti only represented 355 traces, while ZAP got a total of 2.200 traces. This discrepancy also reflects the total storage memory used, since ZAP used 90,3 MB and Wapiti only used 13,1 MB. The memory used, however, is not directly proportional to the number of traces. For instance, in the bWAPP application, the w3af fuzzer generated 5.526 traces, which occupied 523,3 MB. In contrast, ZAP generated 5.947 traces, with little difference in the number of traces, however, the memory used by ZAP is considerably higher than the memory used by w3af (4,9 GB versus 523,3 MB).

Overall, the ensemble fuzzing produced 3.708 traces of DVWA, 8.161 traces of Mutillidae, and 13.388 traces of bWAPP. Hence, in total, the solution analysed 25.257 traces. However, most of these traces were discarded by our solution during the phase of obtaining the interesting traces, resulting in a total of 888 traces that proceeded to the subsequent analysis.

Each scanner was executed sequentially so that identifying the attacks in the traces was more manageable because that traces could only correspond to that specific scanner. Table 5.5 presents the results obtained when inspecting the outputs of the trace analysis tool for each scanner and web application.

Regarding the DVWA application, ZAP is the one that achieved a more significant number of true positives, totalling 21 out of 23, only miss-identifying two requests (detailed below). Wapiti and w3af had

Table 5.4: Number of files, size and traces generated during attack phase by the ensemble fuzzing

	Wapiti-EF			w3af-EF			ZAP-EF		
	Files	Traces	Memory used	Files	Traces	Memory used	Files	Traces	Memory used
DVWA	8	355	13.1 MB	91	1153	52.8 MB	8	2200	90.3 MB
Mutillidae	43	969	1.1 GB	665	4099	5.6 GB	102	3093	3.4 GB
bWAPP	21	1915	146.2 MB	523	5526	523.3 MB	97	5947	4.9 GB

Table 5.5: Analysis of the output of the trace analysis tool

Web App	**Vulnerability**	**Wapiti-EF**	**w3af-EF**	**ZAP-EF**
DVWA	TP	10	10	21
	FN	5	3	0
	FP	0	0	0
	TN	0	1	0
	FFP	0	0	2
Mutillidae	TP	40	79	79
	FN	23	17	8
	FP	0	0	1
	TN	0	0	0
	FFP	32	109	17
bWAPP	TP	3	24	20
	FN	66	28	42
	FP	0	0	0
	TN	0	0	0
	FFP	3	26	221

around half of the true positives compared to ZAP and failed to identify some vulnerabilities, 5 for Wapiti and 3 for w3af. One interesting finding by the w3af scanner was its true negative. This output stated that an attack to a particular URL did not exploit any vulnerability, confirmed in our manual analysis.

For Mutillidae, the number of missed vulnerabilities (FN) increased compared to the previous application, having 23 for Wapiti, 17 for w3af and 8 for ZAP. Regarding the correctly identified vulnerabilities (TP), ZAP and w3af had the highest founds, 79, while the Wapiti scanner only found 40. There is only one case of an FP found for the ZAP scanner where the evaluator could not detect the exploit of the vulnerability. This happened because the input was modified from the entry point until the sensitive sink, and thus making the evaluator stop being able to detect the attack in the trace and miss the attack reaching a sensitive sink.

For bWAPP, Wapiti found very few vulnerabilities regarding the true positives, counting only three true positives and 66 vulnerabilities missed. w3af seems to be the best scanner for this application since

it has found 24 true positives and only 28 false negatives, while ZAP only found 20 and missed 42 vulnerabilities.

In the presented results, however, the sum of the correctly identified vulnerabilities and the missed ones (FN) does not always match in all scanners. A scanner produces multiple results that may correspond to the same vulnerability exploited, meaning that some of the true positives reported may be exploiting the same vulnerability. For instance, in the DVWA case, the scanner from ZAP got 21 true positives, while the total number of vulnerabilities reported was 14 (the sum of all the vulnerabilities regarding the DVWA application in Figure 5.2). This means that the ZAP scanner has produced multiple outputs to multiple requests that exploit the same vulnerability.

Regarding the metrics, Table 5.6 presents the performance results for our solution. Without considering the FFP cases (columns 3 to 6), the results are significantly improved. In general, all fuzzers achieved almost perfect precision in all the applications since the false positives we found were evaluated as false false-positives and were due to the wrong identification of the attacks. The last four columns include the FFP cases and further details are given in Section 5.3.3. In DVWA, ZAP achieves a perfect score in all metrics and is unquestionably the best fuzzer to test this application. In Mutillidae, ZAP seems to also achieve better results than the other fuzzers, having almost 90% of accuracy and above 90% in the remaining metrics. Wapiti and w3af achieve worse results, achieving 64% and 82%,respectively, in accuracy and recall, and 78% and 90% in F-Score. Finally, in bWAPP, Wapiti has under 5% in accuracy and recall due to its high number of false negatives and low true positive rate. ZAP and w3af also do not have astounding results, with values under 50% in accuracy and precision.

The ensemble results could be defined by the best results of the individual fuzzers, however, since it is necessary to analyse the results of all fuzzers in order to assess the fuzzer that achieves higher quality results, we averaged the results of all fuzzers in order to evaluate the whole ensemble. So, our ensemble achieves in all metrics over 81% in DVWA, over 78% in Mutillidae and over 27% in bWAPP. The global results present the average of our solution's performance over the three web applications. With this, our solution achieves nearly perfect precision, with only 1 FP found, and over 62% in accuracy and recall.

The tool's results effectively demonstrate the existence of the vulnerabilities in the source code, since the tool's output presents the trace with which the vulnerability was detected, as presented in Listing 4.18. This trace has information about the values that reached the identified sensitive sinks and it is easy for the user to verify the successfulness of the reported attacks, since the trace demonstrates the behaviour of the application with the attack's payload.

These results reiterate the hypothesis that no scanner achieves better results in all the applications and that the scanners do not explore different applications with different formats in a similar way. Also, we can state that the solution identifies the vulnerabilities associated with the attacks reported by scanners and is able to signalize false positives over the reported attacks. Therefore, we can answer positively to questions (4) and (5).

Table 5.6: Performance test of the solution, for each individual fuzzer, ensemble and global

		Metrics without FFP				Metrics with FFP			
Web App	**Fuzzer**	Accuracy	Precision	Recall	F-Score	Accuracy	Precision	Recall	F-Score
DVWA	Wapiti	0.667	1	0.667	0.8	0.667	1	0.667	0.8
	w3af	0.786	1	0.769	0.869	0.786	1	0.769	0.869
	ZAP	1	1	1	1	0.913	0.913	1	0.955
	Ensemble Average	0.818	1	0.812	0.890	0.789	0.971	0.812	0.875
Mutillidae	Wapiti	0.635	1	0.635	0.777	0.421	0.556	0.635	0.593
	w3af	0.823	1	0.823	0.903	0.385	0.420	0.823	0.556
	ZAP	0.898	0.988	0.908	0.946	0.752	0.814	0.908	0.858
	Ensemble Average	0.785	0.996	0.789	0.875	0.519	0.597	0.789	0.669
bWAPP	Wapiti	0.044	1	0.043	0.082	0.042	0.5	0.043	0.079
	w3af	0.462	1	0.462	0.632	0.308	0.48	0.462	0.471
	ZAP	0.323	1	0.323	0.488	0.071	0.083	0.323	0.132
	Ensemble Average	0.276	1	0.276	0.401	0.140	0.354	0.276	0.227
	Global	0.626	0.999	0.626	0.722	0.483	0.641	0.626	0.590

5.3.3 False false-positive cases

During the experiments, the solution got a large number of results that were originated from the wrong identification of the attacks in the traces. We had to define these cases as false false-positives (**FFP**), since these are the cases where the trace analysis evaluator considered a request to be a false positive, where that request should not even have been considered an attack in the first place. This is due to the evaluator wrongly associating a request with an attack from the report. This happens because it identifies the payload of a successful attack (present in some report) in different traces unrelated to successful attacks. With this, if it identifies an attack in an entry point of a trace that does not exploit a vulnerability, the evaluator will consider that a false positive. It is correct, in a sense, because that payload is not exploiting a vulnerability, and the evaluator detects that and reports it as a false positive; the main problem is the poor identification of the attacks in the traces. These cases are persistent because the payloads sent by the scanners of our chosen set are often similar when attacking multiple endpoints, and the tested applications use similar names in the parameters.

Table 5.5 depicts the number of FFP generated by the solution for each web application (lines 6, 11 and 16). In total, it produced 410 FFP out of 888 traces. For DVWA, the solution only had 2 FFP for ZAP, but the number of these cases increased in the Mutillidae application, having 32 for Wapiti, 109 for w3af and 17 for ZAP. Regarding the bWAPP application, it has the most considerable number of misidentified vulnerabilities compared to the other scanners, with 221 FFP for ZAP, while w3af has 26 and Wapiti only 3.

Table 5.6's last 4 columns present the performance results for our solution considering the false false-positive cases. In this specific case, we counted the false false-positive cases as false positives in order for it to be applied in the metrics. Once again, no fuzzer gets outstanding results in all the web applications.

ZAP got, overall, better performance in DVWA, despite having less precision than the other fuzzers, compensating it by having a recall value of 1. In Mutillidae, ZAP once again got the overall best performance out of all the fuzzers, getting above 75% in all the used metrics. However, in the bWAPP application, ZAP gets the worst results by far, with values under 10% in the accuracy and precision metrics and this is due to the extensive number of false false-positives found by it. In the bWAPP application, the w3af fuzzer seems to achieve better results out of all three fuzzers, despite having all the metric values below 50%.

Compared with the metrics without the FFP, the major difference is the values in precision, that considerably degrade when the FFP are considered. Regarding the ensamble average in DVWA, precision drops to 97%, 60% in Mutillidae and 35% in bWAPP. The global metrics precision is naturally affected and registers 48% in accuracy and around 60% in the remaining metrics. Despite this and the high number of FFP produced by our solution, the global metrics of its performance allows us to still have a positive answer to questions (4) and (5).

5.3.4 Comparison between scanners and trace analysis

Although the considerable amount of false false-positives, these results become more interesting when paired with the data used of Table 5.1 and Figure 5.2. In other words, we compared the results obtained after the trace analysis (for all scanners) with the total number of vulnerabilities found for each scanner, the data obtained in Section 5.2.2. Table 5.7 presents the results obtained from this comparison.

Table 5.7: Comparison of the vulnerability detection in the ensemble version as a whole

Web App	Total reported by Scanners	TP	FN	TN	FN but FP (TN)
DVWA	14	12	1	1	1
Mutillidae	94	77	17	0	2
bWAPP	37	18	19	0	11

Here, the second column represents the total number of vulnerabilities found, which corresponds to the total number of vulnerabilities of the distribution in Figure 5.2. The TP column represents the number of vulnerabilities found by at least one scanner and confirmed by the solution. The FN column represents the number of vulnerabilities missed by our implementation. The TN column represents the number of confirmed false positives, that is, the number of true negatives for our solution. Finally, the FN but FP (TN) column represents the number of vulnerabilities missed by our implementation, but that were, after manual inspection of the scanner results, considered false positives. Here, we consider this specific case to be also a true negative, since the scanners fail to identify the attacks in traces where the attack did not happen successfully.

As we can observe, in DVWA, the fuzzers were able to detect the vast majority of the vulnerabilities

(number of TP), missing only one (FN), that was actually confirmed as a false positive after a manual analysis (FN but FP). Additionally, one of the fuzzers was also able to detect and confirm a false positive in the results reported by the scanners.

In the Mutillidae application, the percentage of the vulnerabilities found by at least one scanner decreased, being only 77 in 94 and no false positives were confirmed in this application.

Finally, in the bWAPP application, the percentage of vulnerabilities found by at least one scanner is even lower (18 out of 37), however, 11 out of the 19 vulnerabilities missed are confirmed false positives.

These results denote that, despite the considerable amount of misidentified attacks, the results of the set of scanners can confirm the majority of the attacks reported in the scanning phase. Additionally, while the number of direct TN (where our solution directly reports that the vulnerabilities reported by the scanners are false positives) is low, in some applications (bWAPP, for instance), the majority of cases where our solutions fail to identify vulnerabilities reported by the scanners (FN), are not vulnerabilities (FP), which makes the non-reporting the right decision, and we consider these results also true negatives. Based on these results, once again, we give a positive answer to questions (4) and (5).

Chapter 6

Conclusions

This dissertation presents a solution for identifying XSS and SQLi vulnerabilities in the source code of web applications written in PHP without inspecting their source code. The solution leverages in an ensemble fuzzing for crawling and scanning the applications under test and uses an web application monitor for detecting and collecting the application's execution traces generated in the scanning phase. Furthermore, the solution correlates the collected traces with the vulnerabilities reported by the fuzzers of the ensemble, identifying the attacks in the traces and validating whether they reached sensitive functions and exploited vulnerabilities. Finally, it presents the vulnerabilities identified by the analysis, as well as its location in the source code of the web application, and the exploits.

The solution was implemented in a tool which comprises three fuzzers and was tested with three web applications. Interesting results showed that our ensemble fuzzing is able to identify vulnerabilities that would otherwise be missed if the fuzzers were to run individually. Furthermore, the solution is also able to confirm false positives in results reported by fuzzers as being vulnerabilities exploited successfully. Despite this, the performance of the solution varies according to the tested applications, which is justified, for one hand in the complexity of the structure of the web applications, and, on the other hand, by the false false-positive results produced by the solution. However, although these latter cases, the solution continues being effective in the detection and identification of web vulnerabilities.

6.1 Future Work

For future work, we believe that the solution should be improved, specially in the correct identification of the attacks in the collected execution traces, before being tested with different fuzzers and web applications. This tuning is essential to lower the number of false false-positives generated during the evaluation phase. One possible solution is the synchronization of the fuzzers with the web application monitor. Although this synchronization could decrease substantially the number of concurrent requests made to the web application, we believe that it could allow to completely eliminate the phase where the interesting

traces are identified, since the traces associated with the attack could immediately be signalized, and no wrong identification could be made.

We currently only consider the XSS and SQLi vulnerability types. For future work, different classes of vulnerabilities could be explored, as well as different combinations of fuzzers and web applications.

Finally, the solution only considers as entry points the PHP Superglobals (e.g., $_GET) and is not able to identify the attack in the traces if the payload changes some mutation from the entry point until the sensitive sink. For future work, we must identify the changes that occurred to the payloads and evaluate if they validate and sanitize the input or if it still remains malicious. With this, the solution will be able to verify the correct sanitization and validation of the inputs in the web application.

References

[1] Blog tool, publishing platform, and cms. URL https://wordpress.org/. 1

[2] Burp suite. URL https://portswigger.net/burp. 12

[3] Owasp top ten vulnerabilities. URL https://owasp.org/Top10. 2, 6

[4] Usage statistics of php for websites. URL https://w3techs.com/technologies/details/pl-php. 1

[5] Google code archive - long-term storage for google code project hosting. URL https://code.google.com/archive/p/skipfish/. 12

[6] Total number of websites. URL https://www.internetlivestats.com/total-number-of-websites/. 1

[7] Vulnerabilities by type. URL https://www.cvedetails.com/vulnerabilities-by-types.php. 2

[8] Open source web application security scanner. URL http://w3af.org/. 12

[9] The web-application vulnerability scanner. URL https://wapiti.sourceforge.io/. 12

[10] Usage statistics of content management systems. URL https://w3techs.com/technologies/overview/content_management. 1

[11] Owasp zap. URL https://www.zaproxy.org/. 12

[12] Iso/iec 27005:2008, May 2011. URL https://www.iso.org/standard/42107.html. 5

[13] Vulnerabilities and exploits, Feb 2016. URL https://www.enisa.europa.eu/topics/csirts-in-europe/glossary/vulnerabilities-and-exploits. 5

[14] Owasp top 10, 2017. URL https://owasp.org/www-project-top-ten/2017/Top_10. 5

[15] The state of web application vulnerabilities in 2017: Imperva, Oct 2019. URL https://www.imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2017/. 5

[16] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/alhuzali. 10

[17] P. Antunes. Monitoring web applications for vulnerability discovery and removal under attack. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, Lisbon, 2018. 17

[18] Archiveddocs. Automated penetration testing with white-box fuzzing. URL https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)?redirectedfrom=MSDN. 12

[19] A. Borges. Bazinga: Whitebox fuzzing for detecting web application vulnerabilities. Master's thesis, Instituto Superior Técnico, Lisbon, 2018. 15

[20] J. Caseirito and I. Medeiros. Finding web application vulnerabilities with an ensemble fuzzing. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pages 19–20, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society. doi: 10.1109/DSN-S52858.2021.00020. URL https://doi.ieeecomputersociety.org/10.1109/DSN-S52858.2021.00020. 4

[21] J. Caseirito. and I. Medeiros. Improving web application vulnerability detection leveraging ensemble fuzzing. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, pages 405–412. INSTICC, SciTePress, 2021. ISBN 978-989-758-508-1. doi: 10.5220/0010484904050412. 4

[22] 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 *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–177, 2016. 16

[23] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang. 14

[24] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, 2004. doi: 10.1109/MSP.2004.111. 10

[25] B. Chess and J. West. *Secure programming with static analysis*. Addison-Wesley, 2007. 10

[26] J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dahse. 10

[27] J. Dahse and T. Holz. Network and distributed system security symposium. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 01 2014. ISBN 1-891562-35-5. doi: 10.14722/ndss.2014.23262. 10

[28] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio. Waf-a-mole: Evading web application firewalls through adversarial machine learning. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1745–1752, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368667. doi: 10.1145/3341105.3373962. URL https://doi.org/10.1145/3341105.3373962. 13

[29] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the USENIX Conference on Security Symposium*, pages 26–26, Aug 2012. 14, 20

[30] K. Drakonakis, S. Ioannidis, and J. Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1953–1970, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417869. URL https://doi.org/10.1145/3372297.3417869. 16

[31] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. 03 2014. doi: 10.1145/2557547.2557550. 12

[32] C. C. Editor. vulnerability - glossary. URL https://csrc.nist.gov/glossary/term/vulnerability. 5

[33] Eevee. URL https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/. 5

[34] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black widow: Blackbox data-driven web scanning. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142, 2021. 14

[35] T. Lee, S. Wi, S. Lee, and S. Son. Fuse: Finding file upload bugs via penetration testing. 01 2020. doi: 10.14722/ndss.2020.23126. 17

[36] I. Medeiros. Web application protection. URL http://awap.sourceforge.net/. 10

[37] I. Medeiros, N. Neves, and M. Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2016. doi: 10.1109/TR.2015.2457411. 10

[38] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2021. 14

[39] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018. doi: 10.1109/TR.2018.2839339. 10

[40] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953. doi: 10.1090/S0002-9947-1953-0053041-6. 9

[41] K. Ryan. Patched zoom exploit: Altering camera settings via remote sql injection, Jun 2020. URL https://medium.com/@keegan.ryan/patched-zoom-exploit-altering-camera-settings-via-remote-sql-injection-4fdf3de8a0d. 2

[42] s0md3v. s0md3v/xsstrike. URL https://github.com/s0md3v/XSStrike. 12

[43] S. Sargsyan, S. Kurmangaleev, J. Hakobyan, M. Mehrabyan, S. Asryan, and H. Movsisyan. Directed fuzzing based on program dynamic instrumentation. In *2019 International Conference on Engineering Technologies and Computer Science (EnT)*, pages 30–33. IEEE, 2019. 15

[44] H. Sohoel, M. Jaatun, and C. Boyd. Owasp top 10 - do startups care? pages 1–8, 06 2018. doi: 10.1109/CyberSecPODS.2018.8560666. 1

[45] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Addison-Wesley, 2007. 11

[46] A. the Author NJ Ouchn ”Passion is needed for any great work. 2013 top security tools as voted by toolswatch.org readers. URL https://toolswatch.org/2013/12/2013-top-security-tools-as-voted-by-toolswatch-org-readers/. 12

[47] M. Vimpari. An evaluation of free fuzzing tools. *Master's Thesis, University of Oulu*, 2015. 13

[48] J. Williams and D. Wichers. OWASP Top 10 - 2017 rcl - the ten most critical web application security risks. Technical report, OWASP Foundation, 2017. 62

[49] W. Xu, S. Park, and T. Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 971–986, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3423340. URL https://doi.org/10.1145/3372297.3423340. 15

[50] J. Zhao and R. Gong. A new framework of security vulnerabilities detection in php web application. In *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 271–276, 2015. doi: 10.1109/IMIS.2015.42. 10

[51] Z. Đurić. Waptt - web application penetration testing tool. *Advances in Electrical and Computer Engineering*, 14:93–102, 02 2014. doi: 10.4316/AECE.2014.01015. 15