UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS

IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

AN AUTOMATED APPROACH TO OFFENSIVE SECURITY

GAETANO PERRONE

TUTOR: PROF. SIMON PIETRO ROMANO

COORDINATOR: PROF. DANIELE RICCIO

XXXIV CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

“Alla mia famiglia, alle persone incontrate e perdute durante questo percorso, a chi ha creduto e continua a credere in me. A Teresa, il mio futuro, che mi spinge a sognare ancora. ”

Acknowledgments

I want to express my gratitude to all the people who have accompanied me on this journey. It has been a journey of obstacles and difficulties, not least the Covid. I thank the Epsilon team, with whom I have grown so much professionally and where I have developed friendships that will probably remain for a long time. I thank Francesco Caturano, a companion of misfortunes and passion for research, which sometimes we betray with that for music! Professor Simon Pietro Romano, who has always transmitted to me all his positive energy in moments of discouragement. I thank Nicola Auricchio and Andrea Cappuccio, whom I hope to have left a sign of my slight madness. I want to thank the NTT Data Security team in Rome and the AI team in Naples that gave me the chance of working on innovative projects with a young and always motivated group. I thank Alessandro Placido Luise for giving a significant boost to Docker Security Playground, our cyber "creature", and Ciro Brandi, who has carried out an excellent thesis despite his work commitments. In general, I would like to thank all the thesis writers that I have followed in these years, who have always shown an untiring desire to learn and improve. Finally, I would like to thank Professor Livio Conti, my "life mentor", who monitors my path from afar, unconditionally, and Master Salvatore Biancardi, an irreplaceable reference point in my personal life.

Gaetano Perrone

Contents

List of Figures

Introduction

Cybersecurity is an increasingly important domain in Information Technology. In a time when each device is connected, cyber threats evolve more and more. Companies need to be protected and to evaluate the potential threats to their systems. There are several approaches to find flaws inside the systems. A very effective one is to simulate the attacker's activities to break inside the environment, obtain access to sensitive information, and compromise the internal network. This kind of activity is called Penetration Testing, and its effectiveness lies in the ability to discover the most critical vulnerabilities. Despite its benefits, companies usually cannot meet their costs, as it requires advanced security experts. Our research work aims to integrate the knowledge of security experts inside an automated system that emulates a Penetration Tester's activities. To accomplish this, we bring three main research contributions:

- We develop behavioural models of Penetration Testing activities;
- We develop a platform that integrates Behavioural Models and implements actions to send attacks;
- We develop several solutions in the so-called *cyber-range* domain to test our platform in realistic virtual environments.

Chapter 1 describes offensive cyber-security concepts that are useful to understand the rest of the work. Chapter 2 gives an introduction to the proposed behavioural models that allows automating Penetration Testing steps. We define Hacker Behaviour in terms of tasks, goals and actions, and then we give a summary description of the realization of an Expert System based on Knowledge Graphs. Chapter 3 describes the proposed automated system framework that is designed with the flexibility of allowing free choice of the most suitable offensive behavioural model to be adopted. Chapter 4 shows our contribution

in the cyber-range domain, in particular the application of hybrid virtualization techniques that leverage microservices-based virtualization without losing anything in terms of vulnerability replication, by also relying on a cloud-based model for the design of cyber-ranges.

Chapter 1

Offensive Cybersecurity

This Chapter gives an introduction to cybersecurity concepts with regard to offensive security methodologies, to the definition of concepts like vulnerability, security control, attack, and to related works. In Section 1.1 we describe offensive security methodologies, particularly in the web application domain. In Section 1.2 we describe useful related security contexts that can help in understanding the proposed work.

1.1 Offensive Security Methodologies

Offensive Security is an approach that allows companies to effectively discover their most critical vulnerabilities. In this Section, we discuss the main Offensive Security Methodologies, as well as several Security Concepts that can be useful to understand the developed framework.

Vulnerability Assessment is a process that aims to find and evaluate vulnerabilities exposed by the analyzed application. It generally relies on the use of automatic tools that are executed periodically. It represents the starting point for prioritizing the vulnerabilities identified in an action plan to enhance and raise the overall security level of an organization. However, the use of automatic tools to carry out the Vulnerability Assessment phase has significant limits; in fact, these activities cannot discover zero-day vulnerabilities, that is, the type of vulnerability previously unknown and for which security patches have not yet been released. Another disadvantage is that they require constant updates (often not foreseen in free releases) to stay aligned with the vulnerabilities identified periodically by security experts.

Penetration Testing (PT) is the process of finding IT security vulnerabilities

in a system. In black-box PT, the team has no information about the target and tries to sneak into the system by exploiting vulnerabilities. An introduction to PT tasks and tools can be found in [1]. Different phases can be identified:

- *Information Gathering*: in this phase the attacker finds useful information that can be used in subsequent phases, e.g., domain names, network infrastructure owned by the target organization, systems that appear to be 'alive' in the network. She/he uses publicly available information to discover sensitive data that might be used to break the systems;
- *Scanning*: in this phase the attacker detects running TCP and UDP services exposed by the target hosts. She/he performs scanning techniques, like "syn scan" or "tcp-full scan" in order to detect open TCP and UDP ports;
- *Enumeration*: in this phase the attacker enumerates running services. The goal here is to detect versions of running services and look for vulnerabilities of exposed services by querying known cybersecurity vulnerability databases (like, e.g., CVE[1]); the attacker can also test the infrastructure through several black-box activities (i.e., fuzzing, Dynamic Attack tools) to discover unknown vulnerabilities;
- *Exploit*: when the attacker has detected vulnerabilities in the system, she/he tries to exploit them and get inside the target;
- *Post-Exploitation*: the attacker tries to obtain higher privileges and persistence inside hacked systems, and performs lateral movement activities to gain access to other internal systems.

Final deliverable of a PT activity is a detailed report, containing an executive summary, i.e., a synthesis of detected vulnerabilities, as well as the list of vulnerabilities ordered by risk level. Each vulnerability contains:

- *a risk level*, that is a combination of different factors (exploitability level, impact, etc.);
- *a Proof of Concept (PoC)*: all detected vulnerabilities should be easily reproducible by the recipient of the report. In such a section, a step-by-step guide is added to describe how to reproduce the attack;

[1] https://cve.mitre.org/

- *recommendations*: countermeasures to be put in place to remove the identified vulnerabilities.

Penetration Testing can be either white-box or black-box. In the white box PT, tests are much more accurate, since in the attack phase the operator can focus better on the specified target and bypass the preliminary information gathering phase because he is already aware of the details of the system and, therefore, test execution times are shorter and reports more detailed. The black box approach represents a scenario closer to a real attack made by an external attacker. It is the exact simulation of a malicious hacker who, from scratch, has to commit himself to discover both the infrastructure and the services exposed by the system. Therefore, the process takes longer and the resulting reports are less detailed than in white box PT, yet more similar to real attack cases. Both scenarios are useful to evaluate the security status of systems, of intrusion detection software that may be present or, more generally, of anything that represents the attack surface. Finally, it must be noted that the two approaches are frequently combined to get a more complete vulnerability analysis.

Nowadays, experienced penetration testers are highly sought job titles. A successful penetration tester is a professional figure who requires not just computer and network security skills, but also intuition, lateral thinking and a lot of experience. In the penetration testing cycle, it is also essential to write a detailed report to the client that shows the services provided, the methodology adopted, as well as testing results and recommendations. The reason why organizations invest more and more in penetration testing, in addition to the requirement to identify vulnerabilities in systems, is the necessity to determine the effectiveness of the security measures that have been taken. Furthermore, marketing reasons must also be considered: obtaining a safety certification by a prestigious penetration test organization contributes to give an added value to a product that can be purchased by third parties. Unfortunately, it must be noted that a security certification does not guarantee the lack of vulnerabilities in the analyzed system but certainly reduces a good part of them. Another fundamental aspect is to execute the tests periodically as the system changes. For example, let's suppose that a test is completed at time t_0 and all the resulting vulnerabilities are fixed at $t_1 > t_0$; a 0-day vulnerability is found at $t_2 > t_1$ and a security patch is released at $t_3 > t_2$. Besides, let's suppose this patch includes some vulnerabilities in a part of the system that previously was considered safe by the pen test completed at t_0. It's clear that the old test is not valid anymore and a new test is needed in order to find the new flaws.

Penetration tests can be divided into the following five categories:

1. **Network Service Tests**: aim to discover vulnerabilities and gaps in the network infrastructure of the clients. Tests focus either on detection of misconfigurations in the target network devices such as firewalls, proxies, routers, DNS servers, or in vulnerable active services such as *fingerd*, a demon that implements the finger protocol. An example of misconfigurations in network devices is a DNS that allows a zone transfer, exposing internal network topology to the attacker.

2. **Web Applications Tests**: aim to secure those applications that can be accessed via web through the network (Intranet or Internet) in a client-server architecture. It is more than just a series of tests; it is rather a more detailed process that must take into account many difficulties such as the differences among web applications (each web application is different from others and may contain unique vulnerabilities).

3. **Client Side Tests**: the goal of these tests is to locate security threats that may be found in client side software such as desktop applications. For example, there could be a flaw in a software application running on the userâs machine which a hacker can exploit. Furthermore, executing uncertified open source software (without testing) may cause sever threats that can't be expected.

4. **Wireless Network Tests**: they aim to evaluate the security of wireless infrastructures that implement protocols of the 802.11 family (access points, hosts) to prevent third parties illegitimate access and Denial of Service (DoS) attacks. There are many automatic tools that may be used such as those of the *Aircrack* suite.

5. **Social Engineering Tests**: they attack the human part of a company. They involve human handling tactics like Dumpster Diving, Phishing, Imitation, Intimidation or techniques for inducing the target, via phone calls, to snatch sensitive data.

In our research work we focused on the automation of Web Application Penetration Testing, for several reasons:

- Trends show a growth of web vulnerabilities. All the companies expose a public website, that usually becomes the main entry point for the attackers.

- As it is complex to model the behaviour of a Penetration Tester, as well as of the target system, we need to decrease the complexity of the problem, so we focused on Web Application Penetration Testing. Anyhow, the results of our research can be extended to any other field of application of Penetration Testing.

1.1.1 Web Application Penetration Testing

When trying to find a methodology for performing a Penetration Test against a Web Application (meaning those accessed using a browser to communicate with a web server), one should keep in mind that Hackersâ activities to find new vulnerabilities always involve a great deal of creativity. It is possible, though, to explore all the regions of the applicationâs attack surface and gain some assurance to have found many issues, according to the available resources.

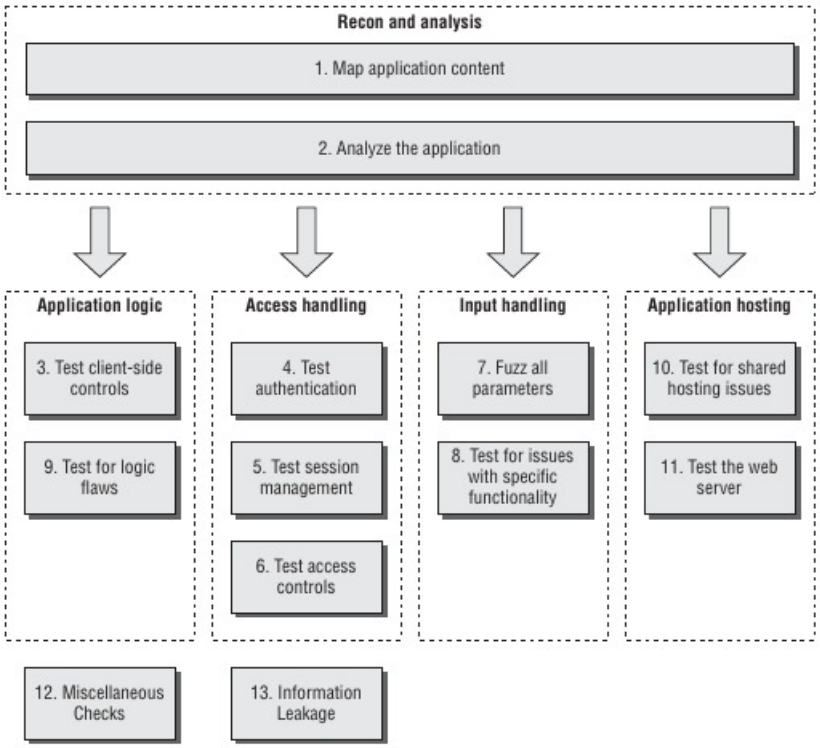

Figure 1.1: WAPT Methodology

Figure 1.1 summarizes the methodology that a web application penetration tester adopts when approaching the study of the target website.

It is divided in two phases:

1. The penetration tester attempts to create a “footprint” of the web application. This includes:
    - Gathering its visible content, exploring public resources as well as discovering information that seems to be hidden. It is also possible, even in this early stage, to identify those application functions that are accessed by passing an identifier of the function in a request parameter;
    - Analyzing the application and identifying its core functions, especially those the web application was designed for. The purpose is to have a map of all the possible Data Entry Points that the application exposes, which are the main flaws that a hacker recognizes in the target application. The penetration tester in this phase should also be able to recognize the technologies that concur to create a core functionality. At the end of this stage, the hacker usually has a clear idea of the path to follow in order to carry out the attacks.
2. In the second phase, the penetration tester knows which road to take and whether to focus on the way the application handles the inputs or on probable flaws in its logic. To have a comprehensive understanding of the applicationâs holes it is however important to explore all of the following areas:
    - Focusing on the application logic means studying the Client-Side Controls to find a way to bypass them. Usually an attacker with minimal skills and equipped with simple tools is enough to circumvent most controls. However, it is important to identify all data being transmitted via the client to understand the validation performed and test how the server responds. A completely different matter is attacking the applicationâs logic, which involves a great amount of lateral thinking. There are some basic tests, which involve removing parameters from requests, using forced browsing to access functions out of sequence and submitting parameters to different locations within the application. How the application responds to these requests can be a sign of some defective behavior that can lead to a malicious effect.

- The stage that involves analyzing how the application handles access to private functionality might be the one to focus on immediately, because authentication and session management techniques are usually full of design and implementation flaws. Attacking authentication can be done systematically, for example checking for bad passwords, ways to find out usernames or vulnerability to brute-force attacks. The session management mechanism is often a rich source of potential vulnerabilities. Its role is to identify the same user across different requests. Breaking this mechanism means jumping into a userâs session. If that user has administrator privileges, this usually allows to compromise the entire application. Less systematic is attacking access controls, because they can manifest themselves in different ways and arise from different sources. In many cases, finding a break in access controls can be done by simply requesting an administrative URL and gain direct access to the associated functionality. In other cases, it can be very hard, mostly because these kinds of errors can derive from deep application logic defects.
- Input Handling attacking techniques are definitely the most well-known, because important categories of vulnerabilities are triggered by unexpected user input. The application can be probed by fuzzing the parameters passed in a request. See the next section for insights on this topic.
- The website can represent an entry point that allows the attacker to have a complete understanding of the targetâs network infrastructure. Defects and oversights within an applicationâs architecture often can enable the tester to escalate an attack, moving from one component to another to eventually compromise the entire application. Shared hosting and ASP-based environments present a new range of difficult security problems, involving trust boundaries that do not arise within a single-hosted application. When attacking an application in a shared context, a key focus of the efforts should be the shared environment itself. One should try to ascertain whether it is possible to compromise that environment from within an individual application, or to leverage one vulnerable application to attack others. Furthermore, the web server represents a significant portion of the attack surface via which an application may be compromised. Defects in an application server can often directly

undermine an applicationâs security by giving access to directory listings, source code for executable pages, sensitive configuration and runtime data, and the ability to bypass input filters.

1.1.2 OWASP Methodology

Over the years, great efforts have been made to try to categorize the security risks that can affect web applications. OWASP (Open Web Application Security Project) is an international foundation that works to improve the security of the web applications. They periodically release a top 10 list of the major risks affecting most of the web applications encountered on the web during the analyzed period, together with issues and concise recommendations on how to mitigate them. Although the original goal of the OWASP Top 10 project was simply to raise awareness amongst developers and managers, it has become the de facto application security standard.

OWASP Top 10 - 2017
A1:2017-Injection
A2:2017-Broken Authentication
A3:2017-Sensitive Data Exposure
A4:2017-XML External Entities (XXE)
A5:2017-Broken Access Control
A6:2017-Security Misconfiguration
A7:2017-Cross-Site Scripting (XSS)
A8:2017-Insecure Deserialization
A9:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring

Figure 1.2: OWASP Top 10 2017

1. **Injection**: is the most dangerous flaw. Injection attacks happen when untrusted data are sent to a code interpreter through an input field or

some other data submission methods to a web application. For example, in SQL injection attacks, an attacker could enter some artfully crafted SQL code into an input field that expects a plaintext username. If that input is not properly 'sanitized', this would result in the execution of that SQL code. Injection attacks can be easily prevented by validating and/or sanitizing user-submitted data. Validation refers to rejecting suspicious-looking data, while sanitization consists in cleaning up the suspicious-looking parts of the data. It is also recommended to use bind variables to manage information coming from outside rather than concatenating it to a statement that is executed server side.

2. **Broken Authentication**: vulnerabilities in Authentication systems usually give attackers access to user accounts and even the ability to compromise an entire system if the attacker manages to access the system as an admin user. For example, an attacker can take a dictionary containing thousands of known username/password combinations obtained during a data breach and use scripts or tools to try all those combinations on a login system to see if there are any that work. Some strategies to mitigate authentication vulnerabilities are limiting or delaying repeated login attempts using rate limiting techniques. Another important security measure is to make sure that session tokens and cookies expire after a specific time interval, otherwise valid sessions can be created by an attacker who can gain access to confidential data.

3. **Sensitive Data Exposure**: happens when web applications donât protect sensitive data such as passwords, banking or user-specific information. If these data are not protected, an attacker can gain access to them and use them for malicious activities. One popular method for stealing sensitive information is by implementing a man-in-the-middle attack. Data exposure risk can be minimized by encrypting all sensitive data as well as disabling the caching of any sensitive information.

4. **XXE** (XML eXternal Entity): this attack affects a web application that parses XML input. This input can reference an external entity, attempting to exploit a vulnerability in the parser. An external entity usually refers to a storage unit, such as a hard drive. A weakly configured XML parser can be tricked into sending data to an unauthorized external entity, which can pass sensitive data directly to the attacker. To prevent XXE attacks, web applications should accept a less complex type of data, such

as JSON, or at the least XML parsers should disable the use of external entities in an XML application.

5. **Broken Access Control**: refers to a system that controls access to information or functionality. Broken access controls allow attackers to bypass authorization and use sensitive functions as if they were privileged users such as administrators. For example a web application could allow a user to change which account they are logged in simply by changing part of a URL, without any other verification. Access controls can be secured by ensuring that a web application uses authorization tokens and sets tight controls on them.

6. **Security Misconfiguration**: includes all those unnecessary features that are included in the web application such as default configurations, development functionality that is mistakenly left in the production environment or displaying excessively verbose errors. For instance, an application could show overly-descriptive errors to users which may reveal vulnerabilities. This can be mitigated by removing any unused features in the code and ensuring that error messages are as generic as possible.

7. **XSS** (Cross Site Scripting): this flaw allows users to add custom code into an URL path or onto a website that will be seen by other users. This attack doesn't target the web application itself but the clients of the application. There are three different kinds of XSS which will be discussed in details later. This vulnerability can be exploited to run malicious JavaScript code on a victimâs browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites. Mitigation strategies for cross-site scripting include escaping HTTP requests input as well as validating and/or sanitizing user-generated content. Using modern web development frameworks like ReactJS and Ruby on Rails also provides some built-in cross-site scripting protection.

8. **Insecure Deserialization**: this threat targets the applications which frequently serialize and deserialize data. Serialization means taking objects from the application code and converting them into a format specified by the application that can be used for other purposes. Deserialization is the reverse process: converting serialized data back into objects the application can use. An insecure deserialization exploit is the result of deserializing data from untrusted sources, and can lead to serious consequences like DDoS attacks and remote code execution attacks. While

countermeasures can be taken to try to catch attackers, such as monitoring deserialization and implementing type checks, the only secure way to protect against insecure deserialization attacks is to forbid the deserialization of data from untrusted sources.

9. **Using Components With Known Vulnerabilities**: web developers often use components such as libraries and frameworks in their web applications. These components are pieces of software that help developers avoid redundant work and provide needed functionality according to the reusability principle. Attackers look for vulnerabilities in these components, which they can then use to carry out attacks. An attacker who finds a security hole in one of these components could leave hundreds of thousands of sites vulnerable to exploit. To minimize the risk of running components with known vulnerabilities, developers should always keep the components updated to the latest security patch.

10. **Insufficient Logging And Monitoring**: data breaches are often discovered when attackers already caused damages to the application. Therefore, OWASP strongly recommends to build sufficient logging and monitoring infrastructure in order to be aware of the attacks made to the applications.

1.2 Security Context

The Internet Engineering Task Force (IETF) provides a rich glossary of definitions applicable to the information security technology field. This information is useful to model the attacker's behaviour and design an offensive automation tool.

According to RFC 4949 [2]:

- **Threat**: A potential for violation of security, which exists when there is an entity, circumstance, capability, action, or event that could cause harm;

- **Flaw**: An error in the design, implementation, or operation of an information system. A flaw may result in a vulnerability;

- **Vulnerability**: A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy;

- **Attack**: An intentional act by which an entity attempts to evade security services and violate the security policy of a system. That is, an actual assault on system security that derives from an intelligent threat;
- **Attack Tree**: A branching, hierarchical data structure that represents a set of potential approaches to achieving an event in which system security is penetrated or compromised in a specified way;

Our work is related to these security concepts: we find a way to discover web vulnerabilities by automating the security tests in Web Applications. To create an automatic system, we need to model an attacker's behaviour, which is strictly related to attacks and attack trees.

1.2.1 Attack Classification and Knowledge Graphs

Attack classification represents a crucial activity in different security areas. During a security assessment, it makes it easier to define which attacks must be performed. When conducting threat modeling activities, it simplifies the definition of attack graphs. Many works have addressed the attack taxonomy problem by introducing different ways to classify attacks. However, these classifications are based on vulnerabilities. They hence assume a 'defensive' perspective. Nowadays, companies have a growing interest in Penetration Testing activities, as they have proven effective in detecting vulnerabilities. Penetration testers perform their activity by focusing on goals rather than attack types. Our contribution in this field is to introduce a "goal-centric" methodology that classifies attacks in terms of Hacking Goals. According to [3], an attack classification can be used to build secure systems, to identify vulnerabilities for which security defences do not yet exist, to provide a uniform language for reporting incidents to response teams. These are all defence perspectives. There is also an offensive perspective that is used to detect vulnerabilities by simulating malicious activities. These activities follow known methodologies, such as those mentioned in [4]. However, in the literature, there are just a few contributions that try and formalize these methodologies.

Many authors have defined methodologies to classify attacks in computer systems. V. M. Igure and R. D. Williams [3] give a formal definition of attack taxonomies and offer a complete overview of the existing ones. In a crucial section of their paper ("Properties of a Taxonomy for Security Assessment"), the authors define basic properties that a taxonomy presents. Among them, an interesting one is "Taxonomy must be layered or hierarchical": authors suggest to create a layered taxonomy providing an objective methodology to identify

vulnerabilities. It is the essential hacking goal classification feature, as by using a goal-centric attack classification, it is possible to focus on hacking goals dependencies. For instance, to perform "Exploit Cross-Site Scripting vulnerabilities", an attacker needs to find one such vulnerability. Thus, the "Exploit Cross-Site Scripting vulnerabilities" goal depends on the related "Detect Cross-Site Scripting vulnerabilities" goal. It is a trivial example, but hacking goal dependencies can be much more intricate than that.

Common Attack Platform and Enumeration (CAPEC) [5] is a community resource for identifying and understanding attacks. It offers a search engine that allows users to search for specific attacks. The classification is advantageous because it reports descriptions and relationships between attacks. It describes prerequisites to perform the attack. The main difference from hacking goal classification is that CAPEC classifies attacks using a target-centric approach, as some prerequisites depend on the target. For instance, CAPEC-66 entry (*SQL Injection*) requires that the application does not correctly validate "User-controllable input as part of SQL queries". Prerequisites in hacking goals are "attacker-centric". in this case, "Attacker has chosen a target and a path, the attacker has sent a valid HTTP request against a target, the attacker has chosen a valid HTTP request parameter".

Kotenko and Doynikova [6] have created a generator of attack scenarios for network security evaluation. This could be one of the possible evolutions of our work since a goal-centric classification simplifies the realization of attack graphs while also defining a test result evaluation. Different authors have explored security testing by leveraging planning models. Obes et al. [7] show how it is possible to create a PDDL (Planning Domain Definition Language) representation of an attack model. PDDL contains interesting properties such as domain definition, action definition, preconditions required to perform an action and its output. Goal-Centric classification can be used to define a hacking methodology, so it is not only focused on PDDL representation. Each goal could be implemented by using different approaches (PDDL, Markov Chains, Reinforcement Learning, etc.).

1.2.2 Security Automation

In literature, there are several approaches that aim at automating the identification of web application vulnerabilities. We identified three main areas that will be further discussed: automated platforms, automated exploit models, intelligent agents for penetration testing.

Automated platforms. In [8], Djuric proposes a modular black box tool,

WAPT, that aims at improving the detection of vulnerabilities, especially those resulting from improper input validation such as SQLi, XSS and Buffer overflow. The tool is composed of a module that crawls the target application to find web pages, a module that identifies the application's entry points, a series of attack generators and analyzers sub-modules that execute attacks towards a target and analyze the results, and a report module that stores and presents the results. XSS attack vectors are taken from a database that contains many attack patterns, while SQLi flaws are detected according to a similarity criterium between the HTML page resulting from the submission of a known valid input and the HTML page resulting from malicious input. Our work shares with the cited one the modular approach, allowing to clearly separate the phases the test goes through, as well as to determine dependencies among them. However, it differs in the way the vulnerabilities are detected. In fact, we introduce a way to integrate open-source tools that implement well known exploit models. Xiong et al. in [9], define a security model that enables functional separation among the phases of a penetration test. This allows to automate the separate steps, as well as represent the knowledge acquired about the system in a clear way, in order to use it for the subsequent phases. The purpose of the work is to integrate penetration testing into the secure software development life cycle. For this reason, the tool relies on penetration testers' experience and falls under the category of grey-box tools, which means holding a partial knowledge of the system under test. Our platform leverages as well a security model which allows to define the tasks performed in a penetration test and present them as functional modules of the architecture. However, in our platform, the security model is shaped after the behavior of a penetration tester. In section 3.1 we will describe such a testing methodology and how it affects the design of the presented platform. Moreover, when facing the design of the automated platform, we assume to have no previous knowledge of the target system, so the approach is categorized as black-box.

Automated exploit models. Many related works tend to introduce automation in the detection process of single vulnerabilities. The ones discussed here refer to known exploit models that adopt the point of view of the penetration tester. Aliero et al. in [10] propose an object-oriented approach to the development of a tool for the detection of SQL injection vulnerabilities. The authors show that applying a black-box strategy for testing allows to improve the detection accuracy, in terms of false positive and false negative results minimization. Some preliminary phases are identified before the application of the automated scanner takes place. Such phases have the purpose to identify prerequisites that

need to be fulfilled, before starting to test the actual vulnerabilities. Aliero et al. recognize that a full crawl of the target website is necessary, in order to check for SQL injection vulnerabilities. The identification of the necessary requirements before the execution of a task, is leveraged in our platform as well. In fact, we introduce the concept of Hacking Task, as the testing activity currently being conducted by the platform. We analyze the dependencies that need to be fulfilled in order to complete a Hacking Task.

1.2.3 Cyber-Ranges

Cyber-ranges are virtual protected environments containing vulnerable systems. They are commonly used to study the performance of security assessment tools [11]. We also have used cyber ranges to evaluate the performance of the proposed automated system. To take advantage of microservices-based virtualization, we explore the application of this kind of technology to realize vulnerable environments. We also try to define a cyber-range model that can help deploy hybrid environments (composed of several virtualization techniques) in the cloud. Different works have explored the degree of realism and the benefits of using cyber-ranges for training purposes. The literature reviewed by Yamin et al. in [12] shows that, despite their popularity, the usage of microservices in cyber ranges platforms is somewhat underrated. In fact, among those reviewed, only the work of Childers et al. in [13] is a first attempt at introducing OpenVZ containers as a means to improve scalability in large scale hacking competitions. Although this work can now be considered outdated, since much has changed in the container technology, the authors already highlighted the core limitations of OS virtualization, especially when reproducing many categories of vulnerabilities. We will address such issues in this work by integrating containers with other virtualization technologies to ensure a high degree of realism in the emulated scenarios.
Alangot et al. in [14] build a scalable and lightweight infrastructure for hacking exercises using Docker containers. The authors also show how relying on OS virtualization entails a performance improvement in memory consumption and CPU usage, compared to standard virtual machines. However, their comparison does not consider the fact that container-based scenarios rule out those vulnerabilities that can not be deployed in Docker containers.
Other works carried out by Chandra et al. in [15] and Pham et al. in [16], propose, as future work, the idea of implementing containers as a means to increase the scalability of their cyber ranges platforms. The author of [17] focuses on leveraging cyber ranges in order to properly perform a security as-

sessment of a distributed system. He stresses that such an approach naturally lends itself to a holistic view of the security assessment procedures since it looks at the target network as a whole rather than as a collection of loosely coupled components. The paper's focus is not on the possibility of leveraging the cyber range for training purposes but rather on using it to reproduce as closely as possible a system under test to spot out its potential vulnerabilities. In our work, we are not focused on replicating large, real-world and complex systems. Rather, we are interested in covering all types of vulnerabilities (kernel-space, user-space, OS-dependent) by taking advantage of different virtualization techniques.

The authors of [18] underline the effectiveness of cyber-range platforms to improve security skills. They claim that cyber training platforms play a crucial role in improving the skills of any protection team, thanks to the fact that they offer the possibility to develop new knowledge through practice. They also observe that the current availability of system emulation and virtualization platforms makes creating cyber training platforms an affordable process in terms of both time and costs.

The work presented in [19] describes an interesting example of how container-based virtualization can also be used in real-world operational environments in order to increase the security level of networking infrastructures. In the mentioned paper, the authors indeed propose to use the Linux Containers (LXC) technology in order to implement a system of *honeypots* which mimic the behaviour of real network services and act as decoys concerning potential attacks. The approach is relevant since it demonstrates how virtualized environments can, in some cases, come to the rescue of real deployments. Since our goal is to reproduce all types of cybersecurity vulnerabilities, LXC technology is not sufficient in our case. We combine it with other technologies in order to cover a broader range of vulnerabilities.

Cyber ranges can be used in many different contexts. As an example, the framework presented in [20] proposes a three phases process aimed at preparing roles for EXCON (EXercise CONtrol) teams. The authors' idea is to enable full-scaled cyber-incident exercises.

The authors of [19] propose a cyber range for the power industry since power grids (or, more generally, energy distribution systems) are potential targets in cyber warfare. They implement a service-oriented resource management framework using OpenStack[2]. The physical architecture is divided into a Management Network and a Business Network. The business network makes

[2] `https://www.openstack.org/`

use of a firewall to manage user access to the service. They also implement an evaluation component that performs load balancing, information logging, and health monitoring operations. Finally, they focus on the main security challenge to be faced, that is the separation between virtual resources and the Internet.

Development methodologies are instead the main focus of [21]. The authors describe a testbed design life cycle and propose a running example whose implementation is based on OpenStack.

Chapter 2

Behavioural Model of an Attacker

An automated system for Penetration Testing needs to coordinate several actions, and there are dependencies among actions. We develop a behavioural attacker model that allows orchestrating our automatic system. To realise the model, we needed an attack classification focused on attacks rather than on defence strategies. We define a new goal-centric attack classification model allowing to define a behavioural model. In a nutshell, we propose an attacker-centric methodology for attacks classification. In Section 2.1 we describe our Hacking Goal Attack classification strategy, and we show how it is possible to combine several Hacking Goals to complete a Hacking Task. In Section 2.2 we illustrate the advantages and the disadvantages of using a Goal-Centric classification. In Section 2.3 we show a practical example of Hacking Goal classification. In Section 2.4 we show how it is possible to use Hacking Goal Attack classification to model an attack Knowledge Graph.

2.1 Hacking Goals

Hacking Goal classification is composed of:

- **Hacking Goal**: a macro goal that the attacker is going to achieve. An attacker targets different hacking tasks to fulfil her/his final goal. Depending on the chosen goal, value and metrics of hacking tasks can change.
- **Hacking Task**: a specific task that an attacker wants to reach. An attacker performs different "hacking attacks" to arrive at this result.

- **Hacking Attack**: a single action that an attacker performs to reach a specific hacking goal: the attacker needs to carry out "actions" against the target.

- **Target Environment**: A description of the target, composed of different entry points (HTTP Requests for a web application target, running services for a network) and vulnerabilities.

- **Hacker Observations**: When the attacker performs Hacking Actions, she/he acquires knowledge about the target environment. For instance, when the attacker makes a tcp scan against a target, she/he "observes" which services are running on that target. As an example, if the system blocks tcp requests, the attacker observes that the target is using an Intrusion Detection System. The expressiveness of hacker Observations depends on the complexity of the chosen Hacking Goal model: it could be "TCP port 22 is open", or "TCP port 22 is open with a confidence level of 99.999%". Hacker Observations are the expression of security controls and vulnerabilities of the target environment, filtered based on the perception of the attacker. Hacker Observations can influence the attacker's behaviour:

  - During fulfillment of a Hacking Goal, observations can influence subsequent Hacking Actions;

  - Each Hacking Goal has an "output". An output influences which Hacking Goals the attacker can perform. Hacking Goal Output is the analysis of the observations gathered during its execution.

There is a hierarchical dependency among these components.

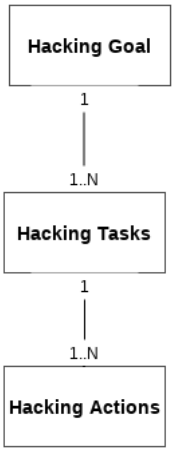

Figure 2.1: Hacking Goal, Hacking Tasks and Hacking Actions relationships

An attacker tries to achieve a hacking goal by executing different hacking tasks. During the execution of a hacking task, the attacker performs hacking attacks to obtain results for the current task. A Hacking Goal could be generic (i.e., 'find all the vulnerabilities in /24 range of the target network') or specific (i.e., 'detect if an attacker is able to gain access to internal software source code by attacking /24 range of the target network'). While a Hacking Goal depends on the current security assessment activity, hacking tasks are independent from the specific activity: a hacking goal guides in the choice of hacking tasks. A hacking task is achieved by carrying out different actions. These actions could be independent from the observations.

2.1.1 Hacking Task Properties

Table 2.1 summarizes the main properties of a Hacking Task, by also providing a short description for each of them.

With respect to Hacking Task metrics, they strongly depend on the specific Hacking Goal the task in question is associated with. Companies might, e.g., be interested in the effectiveness of their attack response strategies. In such a case, they carry out *Red Team* campaigns, that are an evolution of the Penetration Testing activity. While with standard Penetration Testing the target is aware of Penetration Testers attacks and purposefully disables security controls (since there's an interest in having vulnerabilities be disclosed), with Red Team scenarios the attacker needs to evade security controls and thus must

Property Name	Property Description
ID	An identifier. This can either be custom or refer to a standard Security Test classification methodology.
Name	A name that helps understand what is the intent of the current hacking task.
Description	A brief description of the hacking task.
Prerequisites	A list of prerequisites that a hacking task must satisfy in order to be executed. Prerequisites might be the output of a previous hacking task.
Dependencies	The list of hacking tasks that must be completed before the execution of the hacking task in question. For example, before trying an anonymous FTP login, the attacker should detect the presence of a running FTP service inside the system.
Category	A phase of the ongoing security assessment (e.g., Enumeration, Scanning, Exploitation).
Results	Output generated upon completion of a hacking task.
Metrics	A performance indicator that describes how is it possible to evaluate the effectiveness of hacking actions with respect to performing a chosen hacking task.

Table 2.1: Hacking Task properties

necessarily behave in a "stealthy" way. In this case, a Hacking Goal might include "stealthiness" requirements, and the related hacking tasks might assign a higher weight to the actions that do not trigger Intrusion Detection Systems alarms. Hacking Task metrics should in this case include such stealthiness properties.

2.1.2 Hacking Tasks Tree

Hacking task dependencies generate a *Hacking Tasks Tree*.

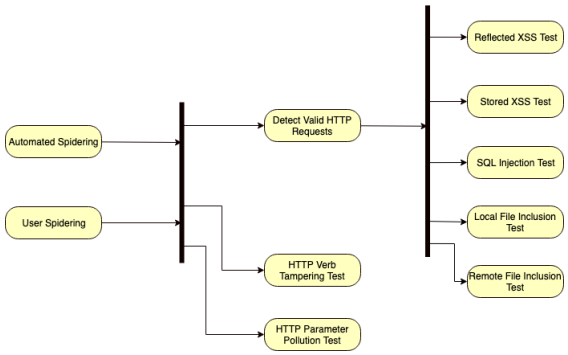

Figure 2.2: Hacking Tasks Tree Example for Web Applications

Fig. 2.3 shows Hacking Task dependencies in a Web Application Penetration Testing model. Each box is a single hacking task. A Hacking Goal in the example is "Find all injection vulnerabilities". An injection vulnerability occurs when a Web Application does not properly validate user input in an HTTP Request. In the example, the Reflected XSS Test detects Cross-Site Scripting vulnerabilities, the SQL Injection Test detects SQL Injection vulnerabilities, the Local File Inclusion Test detects LFI vulnerabilities and the Remote File Inclusion Test detects RFI vulnerabilities.

In order to find an injection vulnerability, the attacker must have chosen a valid path, a valid HTTP request and a parameter of the HTTP request that she/he wants to test. In order to choose a parameter, all forms inside HTML pages have to be found by sending valid HTTP requests to the target. In the model, the "Detect Valid HTTP Requests" hacking task is executed to the purpose. In order to send valid requests, the attacker needs to know available paths at the web server. So, before finding valid HTTP requests, she/he performs the "Automatic Spidering" and "User Spidering" tasks in order to enumerate all paths.

Hacking Goal Classification can use existing sources. As an example, in the case of Web Applications useful resources might be the OWASP [22] (Open Web Application Security Project) Testing Guide and the well-known Web Application Hackers Handbook [23].

2.1.3 Hacking Goal Sources

Hacking Goal Classification can use existing sources. For example, Hacking Goal classification can be used to create a Web Application taxonomy. In this case, used resources to create the taxonomy are:

- OWASP [22] (Open Web Application Security Project) Testing Guide: Open and collaborative Web Penetration Tester methodology;
- Web Application Hackers Handbook [23]: one of the best available resources for web hacker methodologies;
- Penetration Tester experience: security experts providing custom Hacking Goals.

OWASP Mapping

OWASP Testing Project offers a systematic methodology to test web applications. It provides a Testing Guide, containing different "testing controls". Each testing control is identified by an ID having the following syntax: *OTG-CAT-TEST_NUMBER*.

It is possible to map testing controls with the Hacking Goals methodology:

- **ID**: testing controls [22] can be used as Hacking Task ID.
- **Name**: OWASP methodology provides a meaningful name for each testing control (e.g., "Test HTTP Methods"). This can be used as name.

WAH Mapping

An excellent Web Penetration Testing methodology is offered by [23]. Authors introduce a detailed methodology to perform Web Application Penetration Tests, and explore different categories of vulnerabilities, by describing in details "Hack Steps", i.e. hacker actions to be taken in order to detect vulnerabilities.

Hacking Tasks can be identified by using the following syntax: *WAH_CATEGORY_INDEX* where:

- WAH is an identifier used for Hacking Tasks obtained from the Web Application Penetration Testing methodology;
- *Category* is the Hacking Task Category (i.e., mapping, info, etc.);
- *Index* is an incremental number for Hacking Goals that belong to the same category.

Penetration Tester experience

Penetration Tester experience can be used to provide further useful information to model Hacking Tasks.

2.2 Strengths and weaknesses of goal-centric classification

A goal-centric attack classification approach makes it easier to find a mapping with Penetration Testing methodologies, since Penetration Testers use hacking methodologies that are focused on goals rather than on the types of attacks they can perform. Through goal-centric classification it is possible to formalize metrics and evaluate attacks. For instance, if the goal is "Enumerate all Paths of a Web Server", a metric to estimate the effectiveness of performed actions might be the ratio of the number of discovered paths to the number of HTTP requests sent to the Web Server.

The proposed approach might also be used to design intelligent agents. An intelligent agent performs actions inside an environment, and monitors the environment through sensors. It is also important to define agent tasks. Russel [24] defines the concept of "task environment", by using the PEAS (Performance/Environment/Actuators/Sensors) model. In our case, *Performance* refers to the metric used to evaluate the chosen Hacking Goal, *Environment* is the target that the Penetration Tester is analyzing, *Actuators* are the tools and techniques used by the tester and *Sensors* are the "observations" deriving from the executed actions. As part of our ongoing activities, we are formalizing an attacker model based on PEAS, with the aim of showing how it is possible to create a link between Hacking Goal classification and an attacker's behavioral model.

On the downside, the formalization of a goal-centric attack classification model requires proficiency in the security field, as well as specific efforts to

properly define metrics that might change depending on the specific hacking task to be performed.

2.3 Hacking Goal Usage

In this section we illustrate two examples of Hacking Goal classification. In the former example our goal is to find Cross-Site Scripting vulnerabilities in a target website. In the latter, our goal is to gain access to a company's customers information without alerting the company's security staff.

2.3.1 Detect Cross-Site Scripting vulnerabilities

In this example, we suppose to have "hackme.org" as the target web site. Our task is to find all Cross-Site Scripting vulnerabilities it suffers from.

According to [22]:

"Cross-SiteScripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it. An attacker can use XSS to send a malicious script to an unsuspecting user."

1. **Define Hacking Goal**

   The first step to use the methodology is to define a "Hacking Goal". In this example, Hacking Goal is:

   - *Detect all Cross-Site Scripting vulnerabilities in the "hackme.org" target.*

   We make the assumption that the Hacking Task does not include stealthiness requirements, because hackme.org owners want to discover all XSS Vulnerabilities without testing Attack Response strategies.

2. **Define Hacking Tasks**

   Once the Hacking Goal has been defined, we define Hacking Tasks to complete the task:

WAH-MAPPING-001

- **Name:** Automatic Spidering.
- **Description:** The task is to detect the structure of the target by using Automatic Spiders;
- **Prerequisites:** None.
- **HT Dependencies:** None.
- **Category:** Enumeration.
- **HT Output**: The list of Web Pages found on the target site.
- **Metrics**: Total number of HTTP requests sent against the target, total number of Web Pages found.

OTG-INFO-006

- **Name:** Identify application entry points.
- **Description:** Goal is to enumerate attack surface before testing, to identify areas of weakness and detect valid HTTP requests to send Injection Attacks. A valid HTTP request is defined as a request that is answered with response code 20x.
- **Prerequisites:** The user knows the list of URLs.
- **HT Dependencies**: WAH-MAPPING-001;
- **Category:** Enumeration.
- **HT Output**: The list of valid HTTP requests. The list of HTTP requests that contain a value of parameter reflected in the HTTP response page.
- **Metrics:** The list of found valid requests, the list of invalid requests (response code other than 20x), total number of emitted HTTP Requests against the target.

OTG-INPVAL-001

- **Name:** Testing for Reflected Cross site scripting.
- **Description:** The task is to detect Reflected Cross-Site Scripting Vulnerabilities. It is possible by testing each input vector with specially crafted input data, in order to verify if it's possible to inject executable code within an HTTP Response.

- **Prerequisites:** The user has collected the list of valid HTTP requests containing an input reflected in the associated HTTP Response message.
- **HT Dependencies:** WAH-MAPPING-001, OTG-INFO-006
- **Category:** Injection.
- **HT Output:** List of parameters inside HTTP requests that are vulnerable to Cross-Site Scripting.
- **Metrics**: The total number of HTTP Requests sent to the target.

3. **Hacking Tasks Tree**

   Hacking Tasks relate to each other through dependencies.

   Assessor wants to find all Reflected Cross-Site Scripting vulnerabilities (OTG-INPVAL-001). To find vulnerabilities, she/he needs to find Injection entry-points, that are parameters of valid HTTP Requests (OTF-INFO-006). In order to find valid HTTP Requests, she/he needs to discover the structure of the target web site (OTG-MAPPING-001).

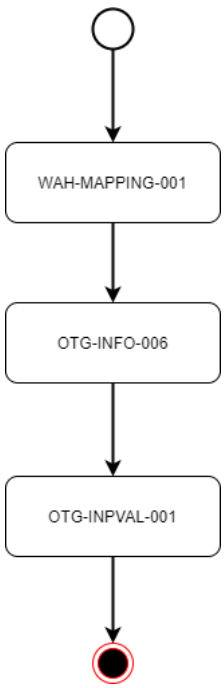

Figure 2.3: Hacking Tasks sequence to detect XSS

2.3.2 Access to customers information without alerting Security Team

In this example, we make the hypothesis that the company "hackme" stores customers information inside an internal database. Our goal is to access such information in a stealthy way, in order to test the effectiveness of the company's Security Operation Center.

1. **Define Hacking Goal**

   We define as "Hacking Goal" the following:

   - *Access customers information stored inside internal hackme databases without being detected by the Security Team.*

2. **Define Hacking Tasks** Hacking Tasks to complete the task are the following:

   PT-STEALTH-WEBHIDE-001

   - **Name:** Create a proxy chains via Tor.
   - **Description:** The goal aims to hide personal IP by using the Tor network.
   - **Prerequisites:** None.
   - **HT Dependencies**: None.
   - **Category:** Stealth.
   - **HT Output**: Anonymized access to the Internet.
   - **Metrics:** None

   PT-ENUM-PASV_SERVICES-001

   - **Name:** Enumerate the system to find vulnerabilities in passive mode.

- **Description:** The task consists in using information gathering tools such as Shodan and Censys to detect and enumerate open services in the target domain. The attacker starts with domain target name, looks for IP subnets and enables enumeration with passive tools, without any interaction with the target. The user also uses tools to create a list of subdomains related to the target.
- **Prerequisites:** The user is connected to the Tor network.
- **HT Dependencies**: None.
- **Category:** Enumeration.
- **HT Output**: A list of open services with version and vulnerabilities.
- **Metrics:** List of actual open services compared to open services detected by solely relying on Shodan and Censys.

PT-AUTH-TESTDEFAULT-001

- **Name:** Test administrator default credentials on exposed public websites that are associated with the target company.
- **Description:** The task is to enumerate attack surface before testing, to identify areas of weakness and detect valid HTTP requests to send Injection Attacks. A valid HTTP request is defined as a request that is answered with response code 20x.
- **Prerequisites:** The user has detected public subdomains; the user has detected open services.
- **HT Dependencies**: PT-ENUM-PASV-SERVICES;
- **Category:** Enumeration;
- **HT Output**: Web applications that use default credentials. Access to administrative interface of web applications.
- **Metrics:** None

PT-PRIVESC-CMDEXEC-001

- **Name:** Privilege Escalation by using administrative web application access.
- **Description:** The task is to perform a privilege escalation by creating web pages that allow to send commands against the target.
- **Prerequisites:** The user knows the list of URLs.
- **HT Dependencies**: WAH-MAPPING-001.
- **Category:** Enumeration.
- **HT Output**: System Access to target machine.
- **Metrics:** None.

PT-POST_EXPLOIT-DATAEXFILTRATION-001

- **Name:** Data Exfiltration;
- **Description:** The task is to exfiltrate data from database. Attacker looks for database credentials used by the application, connects to database, dumps information and downloads it.
- **Prerequisites:** The user has obtained system access to the target.
- **HT Dependencies**: PT-PRIVESC-CMDEXEC-001.
- **Category:** Post Exploit.
- **HT Output**: Customer information.
- **Metrics:** None.

2.4 Beyond Hacking Goals: Attacker Knowledge Graph

Hacking Goals is an approach created to classify the attacker's goals. An application of the proposed classification results in the realization of an attacker knowledge graph. We have tried to extend the concept of behavioural Hacking Goal by implementing an Expert System that uses a Knowledge Graph to suggest the best path that an attacker can follow to find vulnerabilities. We use the OWASP Methodology to create the Knowledge Base. The Knowledge Base is publicly available on Github repository [1]. Any security expert can hence

[1] https://github.com/NS-unina/HackingGoals

contribute in improving it with further information. In this Section, we give a brief introduction to our work. We model the knowledge graph by designing an Entity Relationship Diagram. This will be the core part of the application since data must be compliant to this model to be correctly used. Moreover, this view will be more "database" oriented because, in this way, it will be possible to retrieve data from a database and use them in the built ontology. Figure 2.4 shows the implemented E-R diagram:

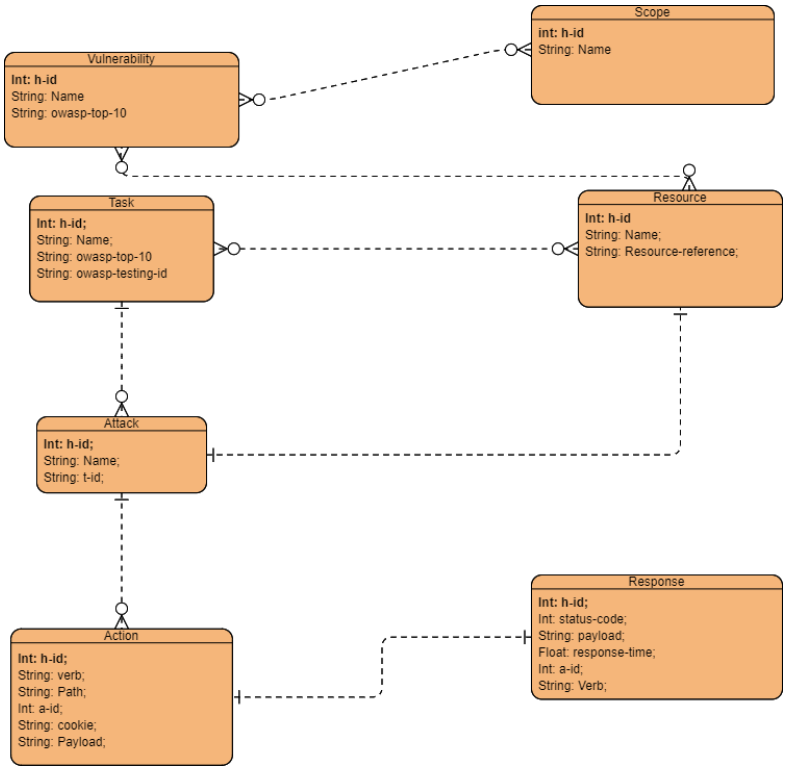

Figure 2.4: E-R Diagram

A detailed explanation of entities and their attributes is provided below:

- **Vulnerability**: "an occurrence of a weakness (or multiple weaknesses) within a product, in which the weakness can be used by a party to cause the product to modify or access unintended data, interrupt proper execution, or perform incorrect actions that were not specifically granted to the party who uses the weakness." [25]. This entity has been modelled

following the CWE definition. The unique identifier of a vulnerability is an id (or an "Hacking-id", also "h-id"), which, for each vulnerability, can be directly obtained from the CWE website and must be an integer. The Name is a string attribute, which gives a short explanation of the vulnerability, e.g. "Improper Neutralization of Special Elements used in a Command". The owasp-top-10 attribute is the vulnerability category, e.g. the previously cited vulnerability belongs to the "Injection" category. Thus, it is possible to search vulnerabilities based on their id, or search for a group of vulnerabilities that belong to the same category. A vulnerability has a many-to-many relationship with the entities "Scope" and "Resource", for several reasons:

  - a vulnerability can violate multiple "Scopes" (e.g. the "Improper Neutralization of Special Elements used in a Command" violates "Integrity", "Confidentiality" and "Availability");
  - a vulnerability can use more than one resource (e.g. "OS Command Injection" can exploit the "Operating System version" and the "Webserver ports" resources);

- **Scope**: The Scope identifies the "application security area that is violated" [25]. The "h-id" is the unique identifier for each instance of the entity and the name gives a short explanation of the scope. The CWE identifies five different scopes: "integrity", "confidentiality", "availability", "Non-Repudiation", "Access Control". Scope has a many-to-many relationship with Vulnerability. In fact, a Scope can be violated by multiple vulnerabilities; e.g., the "Integrity" can be violated by "OS command injection" and "SQL Injection".

- **Resource**: A Resource is any web applicationâs entity that can be either used or discovered by an action performed on the application itself. It can be considered as an application's asset, whose knowledge can be either provided by a task or used to exploit a vulnerability. The unique identifier is the "h-id", and the "Name" briefly describes the resource. The "Resource-Reference" is the category the resource belongs to, and allows for coarse-grained research. For example, a resource named "Archived documents" has the "resource reference" field as "Sensitive information"; another resource, named "Account passwords and other credentials", has the same "resource reference" value. Thus, it is possible to search for a vulnerability linked to all "sensitive information"

resources. The entity Resource has a many-to-many relationship with Vulnerability and Task; it has a one-to-one relationship with the Attack Entity. A Resource can be exploited by different Vulnerabilities; a resource can be used by many different tasks, or knowledge about a resource can be provided/required by several tasks. For example:

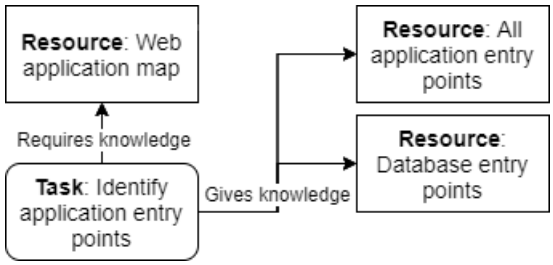

Figure 2.5: Knowledge relationship with resource example

- **Task**: A task is the highest-level type of action that can be performed on the target. It may be the penetration tester's final target (the "Goal" task). In order to execute the "Hacking Goal", the penetration tester has to perform preliminary tasks, hence there are dependence relationships among them, which will be better described in the "Relation Diagram" paragraph. Examples of tasks are:

  1. **Fingerprint Web Server**: try to understand the operating system that runs on the Web Server;
  2. **Testing for Reflected Cross site scripting**: perform a test to assess if the Cross site scripting vulnerability is present;
  3. **Identify application entry points**: identify the application's parameters used in HTTP requests.

  The attributes are:

  – ***h-id***: unique identifier;
  – ***Name***: a brief description of the task;
  – ***owasp-top-10***: specifies the category the task belongs to (e.g., "Testing for Reflected Cross site scripting" has owasp-top-10 category "Injection"), allowing for the coarse-grained research of tasks (e.g., search for all "Injection" tasks);

– ***owasp-testing-id***: is used in compliance to the OWASP Testing Guide, which sets a specific ID for each task (e.g., the task "Testing for Reflected Cross site scripting" has "OTG-INPVAL-001" as "owasp-testing-id"), allowing to use the standardized methodology proposed in it.

A task (or Hacking Task) is composed of multiple attacks and is linked to multiple resources. There may be different links among tasks and resources. An **acquired knowledge** relationship occurs when a task provides knowledge about a resource. For example:

1. the knowledge of "Operating System" resource: our system knows the type of Operating System that is running on the target;
2. the knowledge of "FTP Area" resource: our system knows that FTP service exists in the target application and whether it is reachable or not, which files are contained, etc.;

A **required knowledge** relationship occurs when a task requires the knowledge of a specific resource to be completed. Here is an example of a task composed of multiple attacks:

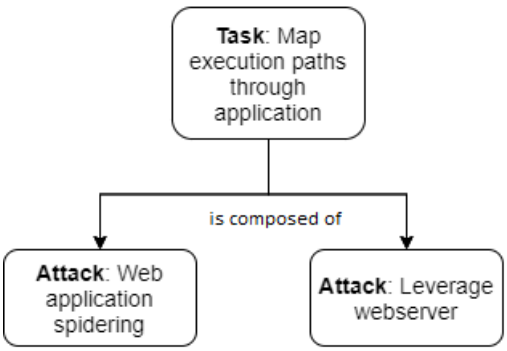

Figure 2.6: Composition of a Task

- **Attack**: An attack is a specific set of actions that can be performed on the target; for example, "Web application spidering" is an attack that includes many actions; it mainly represents a command that must be executed by a tool. For example, the "OWASP ZAP" tool can perform the spidering of a web application, which consists of many HTTP requests sent to the web application; each request has its specific parameters that

must be set by the tool. The "h-id" parameter is a unique identifier, and the Name gives a brief description of the attack. The "t-id" parameter is used to link an attack to its specific task, and contains the "h-id" of the Task the Attack refers to. In this way, it is possible to adopt a standard methodology, building its own set of tasks. We have composed tasks using the methodology given by the OWASP testing guide. The Attack entity has a one-to-many relationship with the Action entity since more actions compose a single attack. The attack entity also has a one-to-one relationship with the Resource entity. When an Attack is successfully carried on the system, it gives the knowledge about one of the resources twith which he corresponding Task has a relationship of acquired knowledge.

- **Action**: An action is atomic, meaning that it is the smallest operation carried on the system. It has the following parameters:
  - *"h-id"*: unique identifier;
  - *"verb"*: it is the request method of the communication protocol which the system uses (for web applications may be GET, POST, PUT, etc.);
  - *"path"*: specifies the path the action has been carried on (it is the URL in the web application scenario);
  - *"a-id"*: it contains the "h-id" the action is linked with;
  - *"cookie"*: represents the cookie contained inside the action (it is in a general form, independently from the used communication protocol);
  - *"payload"*: it is the content of the action and can also be empty (e.g. in "GET" requests).

  The Action entity is linked via a one-to-many relationship with the Attack entity, because an action is specific to a particular attack, and multiple actions compose an Attack. An action is linked to its response. In fact, there is a one-to-one relationship with the entity Response since each action has its own response (and vice-versa).

- **Response**: The entity Response is specific to one instance of the entity Action (one-to-one relationship), and has the following parameters:
  - *"h-id"*: unique identifier;

– *"status-code"*: it specifies whether the action has been successful or not, and possibly why the action has been denied;
– *"payload"*: the optional payload that the system returned;
– *"response-time"*: the timestamp of the received response (may be useful to detect time-based attacks);
– *"a-id"*: it contains the h-id of the action the response is linked to;
– *"verb"*: it is the so-called "reason phrase", used to emphasize the response result (according to the used communication protocol).

2.4.1 Relation diagram

The diagram in Figure 2.7 highlights the relations among the previously listed entities. Relations play a crucial role in an ontology aimed at writing rules that enable the inference process by linking two or more entities. Each entity plays a specific role.

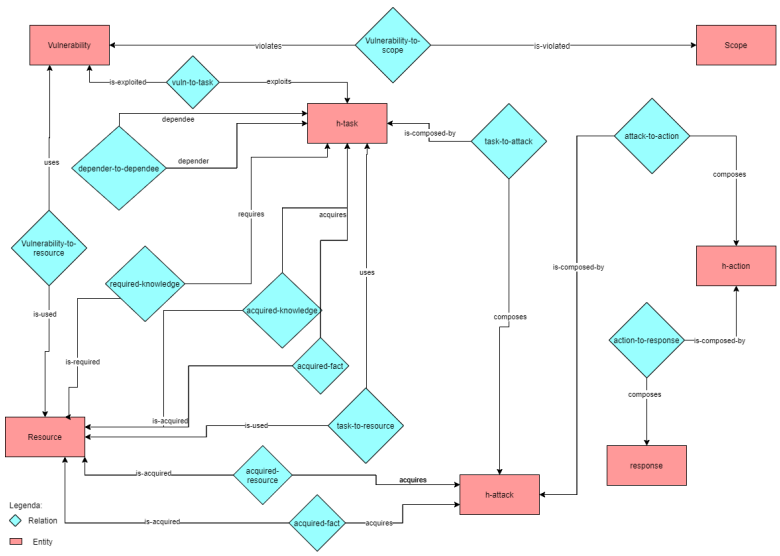

Figure 2.7: Relation Diagram

- **Vulnerability-to-Scope**: relationship among Vulnerability and Scope entity. A vulnerability can violate one or more Scope, and one or more Vulnerabilities violate a Scope. This relationship allows us to look for

a vulnerability that violates a specific scope and, possibly, find the task linked to that specific vulnerability. This kind of relationship should be retrieved from stored data [25], rather than being created with some inference rules.

- **Vulnerability-to-Task**: relationship among a Vulnerability and a Hacking-Task (also H-Task) entity. This relationship is used to retrieve a task linked to a vulnerability in some way (e.g., a vulnerability and a task use the same resource; hence, they are related). This relationship could be directly generated by the inference engine thanks to the rules.

- **Vulnerability-to-Resource**: relationship among Vulnerability and Resource entity. A Vulnerability uses a Resource and one or more Vulnerabilities can use a Resource. This kind of relationship should be retrieved from stored data [25], rather than being created with some inference rules.

- **Depender-to-Dependee**: relationship among two H-Task entities. This relationship allows the creation of an H-Task chain composed of related tasks. For example, an H-Task needs the knowledge about a Resource that another H-Task gives. The relations are generated through the usage of Rules. This relationship is fundamental when the penetration tester sets a "Goal" Task to reach: the system can calculate the chain of tasks that ends with the chosen one task and, going backwards in the chain, it can suggest the first task that the penetration tester must execute. The chain of tasks is updated each time the penetration tester executes the first task, allowing the system to "guide" the penetration tester to the execution of the final "Goal" task.

- **Required-Knowledge**: relationship between an H-Task and a Resource entity. In order to be executed, an H-Task requires the knowledge of a resource (e.g., the H-Task "Identify application entry points" requires the knowledge of the Resource "Web application map"). The task can be executed only when the related knowledge is acquired. This kind of relationship should be retrieved by studying the covered Tasks. We model these relations with the OTG methodology [26]).

- **Acquired-Knowledge**: relationship among an H-Task and a Resource entity. When it is executed, a task provides the knowledge about a Resource; in this way, it is possible to calculate the chain of tasks to execute

the final "Goal" task. This relationship only highlights that a task, when completed, gives the knowledge about a Resource. Information about the completion of the task is required to suggest the next task to execute in the task chain.

- **Acquired-Fact (H-Task)**: relationship among an H-Task and a Resource. This relationship states that the H-Task has been effectively executed. Thus the knowledge of the related Resource has been acquired. In this way, it is possible to advance in the H-Task chain, suggesting the next H-Task that must be executed (to execute the final "Goal" H-Task).

- **Task-to-Resource**: it is a relationship between an H-Task and a Resource entity, which states that an H-Task uses a specific Resource entity to be completed. This relationship can be used to link an H-Task to a Vulnerability through a rule (if the H-Task and the Vulnerability use the same Resource, then create the relationship Vulnerability-to-Task).

- **Task-to-Attack**: relationship among an H-Task and an H-Attack entity. As stated before, an H-Task is composed of more H-Attacks. These relations are formed by studying the specific used tasks [26].

- **Acquired-Resource**: relationship among an H-Attack and a Resource entity. An H-Attack acquires knowledge about a Resource when it is executed. The specific Resource is one of the Resources in the pool of H-Task Resources linked to the H-Attack.

- **Acquired-Fact (H-Attack)**: relationship among a Resource and an H-Attack entity. This relationship is used to state the completion of an attack and the knowledge acquisition of the related Resource.

- **Attack-to-Action**: relationship among an H-Attack and an H-Action entity. As stated previously, an Attack is composed of multiple Actions.

- **Action-to-Response**: relationship between an H-Action and a Response entity. Each H-Action has its own Response.

2.4.2 Hacking Goal

It is essential to define what a "Hacking Goal" is in our context: it is the final aim that the penetration tester wants to reach during his/her activity. It can be a task to be executed, as "Testing for Local file inclusion", or it can

be a particular set of queries to "Find all tasks related to vulnerabilities that violate the integrity scope". It is clear that, based on the "Hacking Goal", the system has to behave differently. In the first example, the system will perform different queries to find the task chain and suggest the task (and attacks) that can be executed; in the second one, the system will give back to the penetration tester a specific set of tasks.

2.4.3 Relevant Rules

The following rules are the most relevant ones for our model:

Attack acquires knowledge

This rule is implemented in order to form the relationship Acquired-Fact between an H-Attack and a Resource instance. The following is a flow chart explanation of the rule: The rule is formally defined as follows:

(AT1, is, H-Attack) $\wedge(AC1, is, H - Action) \wedge (R, is, Resource) \wedge (AT1, attack - to - action, AC1) \wedge (AT1, acquired - knowledge, R) - > (AT1, acquired - fact, R)$

Task Acquires Knowledge

This rule is implemented in order to form the relationship Acquired-Fact among an H-Task and a Resource instance:

(T1, is, H-Task) $\wedge(AT1, is, H - Attack) \wedge (R, is, Resource) \wedge (T1, task - to - attack, AT1) \wedge (T1, acquired - knowledge, R) \wedge (AT1, acquired - fact, R) \rightarrow (T1, acquired - fact, R)$

Depender to Dependee relation

This rule is implemented in order to form the relationship "depender-to-dependee" among two H-tasks instances:

(T1, is, H-Task) $\wedge(T2, is, H - Task) \wedge (R, is, Resource) \wedge (T1, required - knowledge, R) \wedge (T2, acquired - knowledge, R) \wedge !(T2, acquired - fact, R) - > (T2, depender - to - dependee, T1)$

Vulnerability to Task relation

This rule is implemented in order to form the relationship "vuln-to-task" among an H-task and a Vulnerability instance:

(R, is, Resource) $\wedge(T, is, H-Task) \wedge (T, owasp-top-ten, owasp1) \wedge (V, is, Vulnerability) \wedge (V, owasp-top-10, owasp2) \wedge (T, task-to-resource, R) \wedge (V, vulnerability-to-resource, R) \wedge (owasp1, isEqualTo, owasp2) -> (V, vuln-to-task, T)$

Chapter 3

An automated approach to Pentest

In this Chapter, we present the design of an architecture for automated web application penetration testing.
The main contributions of this part of the thesis are the following:

- refine a methodology for Web Application Penetration Testing (WAPT), based on the orchestration of black-box testing units, called Hacking Goals;
- design and implement a distributed platform to automate such methodology, whose main components are an operating unit that performs attacks, called Executor and a control unit that orchestrates them among consecutive phases, called Orchestrator;
- define a flexible way to integrate external tools to implement the Hacking Goals;
- show an integration example to discover cross-site scripting vulnerabilities;

3.1 Design

In this Section, we will describe the design of an automated platform for web application penetration testing. We first summarize a methodology for WAPT and then outline the aspects that are reflected in the architecture.

3.1.1 Web Application penetration testing methodology

WAPT final objective is a comprehensive understanding of the security level of a system under test, measured in terms of vulnerabilities exposed and the indication of their severity.
Penetration testers perform several tasks to accomplish this goal. These tasks help the pentester increase their knowledge about the target system and hence discover vulnerabilities. For instance, enumerating the attack surface of a web application outputs the list of available entry points. Such entry points will be helpful in subsequent phases when the pentester is testing, for instance, injection-based vulnerabilities. We call these "*Hacking Tasks*".
A Hacking Task is a black-box testing activity that probes one or multiple resources of a Web Application. Its purpose is to increase the notion of vulnerability that penetration testers hold about the resources under test.

It is possible to implement each Hacking Task in several ways. Penetration testers often use either open source or commercial tools (or a combination of the above) that help accomplish the objective of each Task.
When a tool is executed against a target, it performs attacks that might harm the System under test. For this reason, a tool used to implement a Hacking Task is considered as a separate attack.
The relation between Attacks and Hacking Tasks is in the form of a loose aggregation. There is no direct correspondence between the two: a Hacking Task might be carried out running either a single or multiple Attacks.
According to Security Experts' experience, a well-established way to arrange Hacking Task tasks into consecutive phases is the following:

1. **Information Gathering**: gain as much information as possible about the target. In the case of web applications, the objective of this Task is to analyze the attack surface and look for the available entry points;

2. **Enumeration**: identify any potential security weaknesses that could allow an outside attacker to gain access to the environment or technology being tested;

3. **Exploitation**: apply specific exploit models, after having reviewed the results of the previous phases;

4. **Reporting**: develop documentation that describes how to reproduce the steps that lead to the discovery of the vulnerabilities.

A relation of dependency exists between Hacking Tasks that belong to consecutive phases. For instance, tasks that perform Enumeration are executed after

the tasks that belong to the Information Gathering area. This allows concatenating tasks, ensuring that the previous Task's output is presented as input to the next one. When users wish to perform a full penetration test, this mechanism is leveraged to perform automated execution of subsequent classes of tasks. It is important to underline that the specific communication protocol is "task-logic" independent: it is possible to change the sequence of tasks completely.

3.1.2 High level architecture

Based on the analysis outlined in the previous paragraph, penetration testers' primary ability is to orchestrate tools throughout several phases to complete Hacking Tasks. To automate such a process, we propose the architecture depicted in Figure 3.1.

- the **Executor** holds the implementation of the attacks. It awaits instructions from the Orchestrator for the specific Attack to run. It also reports back to the Orchestrator the results of Attacks;
- the **Orchestrator** tells the Executor which Attack to set off against a target. It organizes Attacks into Hacking Tasks and determines the sequence of tasks to carry out penetration tests. It arranges the results of the attacks to be presented to users;
- the **Front-End** allows users to initiate testing sessions. Users decide whether the platform should execute a single Attack, a Hacking Task or an entire set of penetration tests.

3.1.3 Offensive Web Application Data Model

Figure 3.2 shows the domain model.

Below are some considerations:

- A target has a base URL and several Urls (i.e. relative links, also called Paths); each Path can be found by sending an HTTP Request;
- HttpInteraction is a couple composed of an HTTP Request and the corresponding HTTP Response. HTTP Attacks can send HTTP Requests against the target and obtain HTTP Responses;

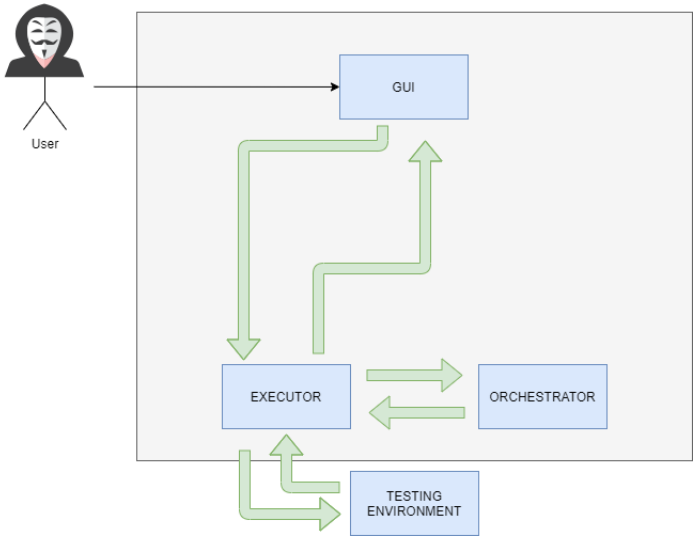

Figure 3.1: High-level system architecture

- A Hacking Task depends on other Hacking Tasks, and it is composed of several Attacks;
- Each Attack is linked with an Analyzer: it takes information about the output of the executed Attack and parses observations;
- Analyzer generates a set of Observations that can be sent to an Anomaly Checker to update the target's knowledge and detect anomalies and vulnerabilities (See Section 3.2.2).

A Hacking Task can be composed of one or more attacks, so it is possible to integrate simple Web Attack tools or combine them to create more complex attack logic. We refer to **Attack Integration** when the Hacking Task is performed through a single Web Attack (for example, by integrating a single web attack tool or by implementing a Fuzz Attack Module). We refer to **Behavioural Model Integration** when the Hacking Task is performed by combining several Web Attacks.

The more penetration testers get deep into the analysis of the System under test, the more they refine the knowledge of its vulnerability status. For this reason, tasks conducted in the early stages of testing are set off against all the available application paths. Instead, towards the end of the activity, single web

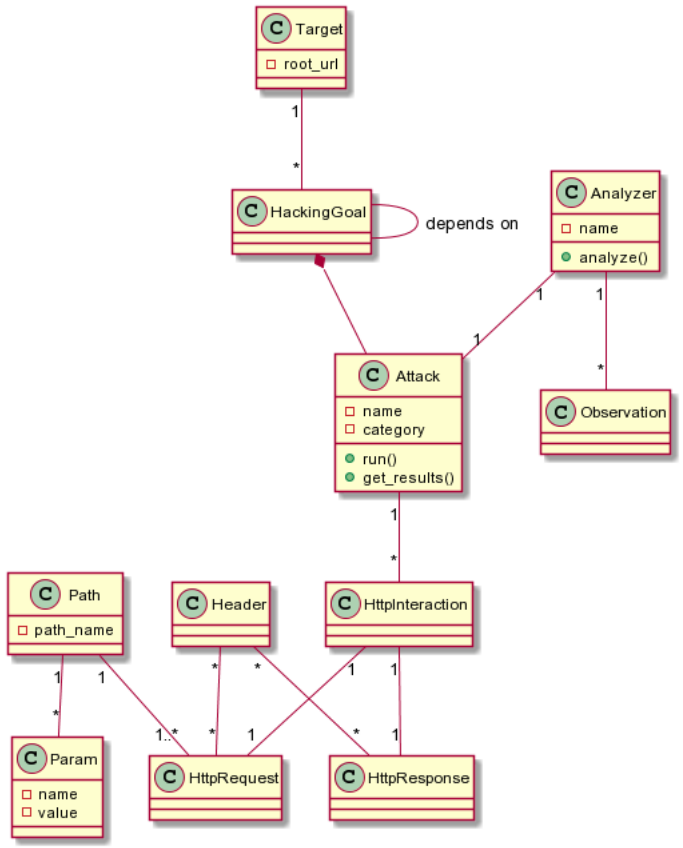

Figure 3.2: Domain Model

pages are tested. To handle this range, Hacking Attacks have been organized in the following categories:

- ***WebServerAttack(base_domain_url)***: Attacks are sent against the Web-Server base domain URL;
- ***PathAttack(path)***: Attacks are performed towards the provided path;
- ***ParameterAttacks(path, parameter)***: Attacks are sent against an input parameter for the provided path.

3.1.4 Orchestrator

The Orchestrator, depending on the Hacking Task category, provides instructions to the Executor to run the corresponding Attacks towards the provided target. The functionality required by the Orchestrator correspond to the principal activities conducted by a penetration tester:

- **Information Gathering**: gain as much information as possible about the target. In the case of web applications, the objective of this Task is to analyze the surface attack, looking for the available entry points;
- **Vulnerability Assessment**: identify any potential security weaknesses that could allow an outside intruder to gain access to the environment or technology being tested;
- **Exploitation**: apply specific exploit models, after having reviewed the results of the previous phases.

The Orchestrator suggests to the Executor the best attacks depending on the current environment state, so it provides a **"Suggest Next Attack"** endpoint that can be used to ask which action should be performed. Vulnerability Detection is a complex task that depends on several factors, such as Response Time, HTML Response Content, etc. It also depends on the acquired knowledge of the System Under Test. For this reason, this analysis is performed by the Orchestrator. The Executor interacts with the Target and sends Observations to the Orchestrator that analyzes them depending on its Behavioural Models. The Orchestrator uses an **Anomaly Checker** module to decide if there is an anomaly or a vulnerability. Finally, the Orchestrator contains Modules that might leverage Artificial Intelligence engines. For example, the Orchestrator can use these engines to parse HTML responses and discover valid HTTP Requests: a form containing a First Name and a Last Name, should generate a request that contains two parameters with alphabetic characters. This Task is performed through an **HTML Analyzer** module developed through a Natural Language Processing algorithm. The Design Decision to keep behavioural and AI models (Orchestrator modules) separated from the found vulnerabilities and the interactions (Executor modules) is due to several reasons:

- **Executor Performance**: the Executor sends attacks against the target and obtains attack outputs. It demands high performance, so it is advisable to keep it as simple as possible;

- **Models Intellectual Property preservation**: it is possible to separate and remotely distribute Orchestrator and Executor; in this way, it is possible to preserve all the information about the behavioural models;
- **Security Issue privacy**: it is possible to deploy the Executor inside the customer environment and the Orchestrator separately. Found vulnerabilities will be stored in the customer environment and protected by privacy issues.

3.2 Communication Protocol

In this Section, we describe the Communication Protocol between the Executor and the Orchestrator.

When users ask the platform to perform tasks, the Executor alone cannot decide which attacks need to be set off against a target. Therefore, it asks the Orchestrator for instructions. A protocol is defined in order to regulate the communication between the two modules.

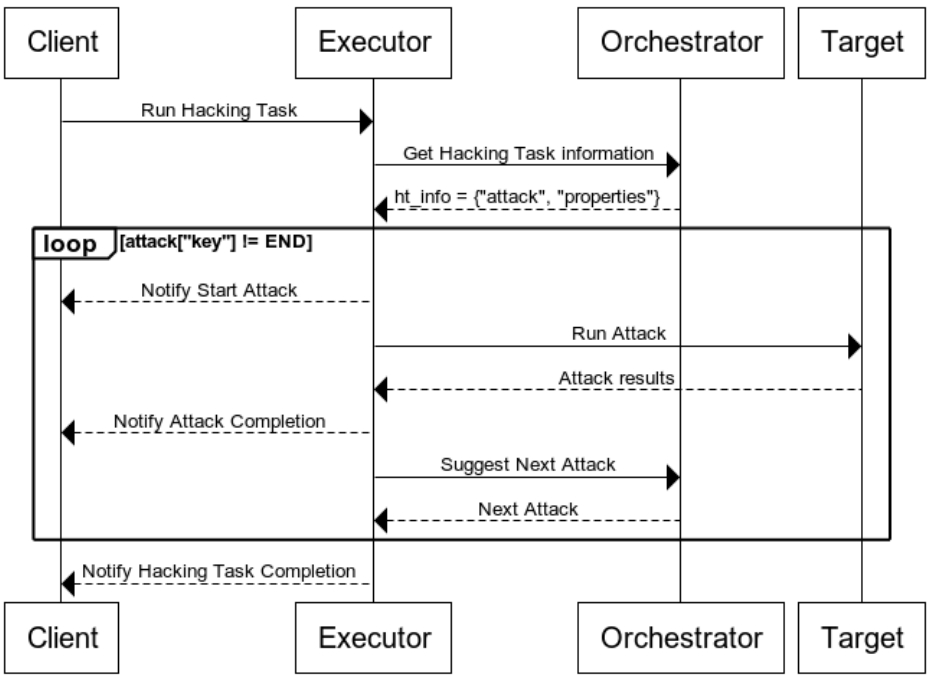

Figure 3.3: Hacking Task Communication Protocol

Figure 3.3 shows the exchange of messages between the main modules to perform a given Hacking Task. Users choose the HT that the automated platform will execute on their behalf. Then, the Client asks the Executor to run the HT. To start the interaction, the Executor asks the Orchestrator for instructions, sending a "*Get Hacking Task Information*" message. The Orchestrator sends information about the given Hacking Task, using a "*ht_info*" message. Such message carries an object called *Attack* that identifies the specific Attack the Executor will run. Then, a loop is repeated until there are no further attacks to run, within the same Hacking Task. A "Suggest Next Attack" message is sent by the Executor to ask the Orchestrator whether there are other Attacks to run before ending the current Hacking Task. The Orchestrator responds with a "*Next Attack*" message, which carries the *attack* object. If all the Attacks have been performed, the *Attack* identifier is equal to "END", which signifies the current Hacking Task is completed. If users wish to perform the entire penetration test, the process described in Figure 3.3 is repeated for every available Hacking Task.

3.2.1 Suggest Next Attack

When the Executor performs an attack, it obtains observations from the target and asks the Orchestrator for the next Attack to be performed. Orchestrator uses Observations to update its information about the target and to choose the next action.

Listings 1 and 2 show Executor Next Attack message and Orchestrator's Response

```
{
    "hg_id": string,
    "action_name": string,
    "current_state": string,
    "observations": { "key_obs":
    ObervationInstance, ...}
}
```

Listing 1: Orchestrator Request

3.2.2 Anomaly Checker

The main goal of the platform is to discover vulnerabilities. After that the attacks are performed, the platform should verify the presence of vulnerabil-

Key	Description
hg_id	A string identifying the Hacking Task that is executed
attack_name	A string identifying the attack just performed. NULL in suggest_first messages
current_state	A string identifying the current_state. NULL in suggest_first messages and in those actions, which do not require states evolution
observations	A json object containing the observations obtained by analyzing the responses to the action just performed. Its structure changes as the action changes. NULL in suggest_first messages and in those actions, which do not need to obtain observations

Table 3.1: Suggest Next Attack parameters

```
{
        "key": "next_action",
        "data": {
                "next_action": string,
                "next_state": string,
                "action_properties": {...},
}
}
```

Listing 2: Orchestrator Response

ities. The Executor asks the Orchestrator to check for anomalies through the *Anomaly Checker Request* (Listing 3). The Orchestrator analyzes the observations acquired during the attacks and replies with a verdict (Listing 4).

```
{
        "key": "anomaly_check_output",
        "data": {
            "output": string
        }
}
```

Listing 3: Anomaly Checker Request

Key	Description
key	A constant identifier
data	A JSON object containing the effective message
next_attack	A string identifying the next attack suggested by Orchestrator. "END" means no further actions are required.
next_state	A string identifying the next state the Executor will assume during Hacking Task
attack_properties	A JSON Object whose structure depends on next_attack. *NULL* if it is the *END* action.

Table 3.2: Suggest Next Attack parameters

```
{
        "key": "next_action",
        "data": {
                "next_action": string,
                "next_state": string,
                "action_properties": {...},
}
}
```

Listing 4: Anomaly Checker Response

The Anomaly Checker Response can be:

- **Normal**: there is no presence of vulnerability;
- **Anomaly**: there is an anomaly that cannot be considered a vulnerability, for example an error message;
- **Vulnerability**: the platform has detected a vulnerability

As described above, there are three types of attacks, each one is related with a domain entity. Depending on the performed attack, the anomaly status is associated with the related domain entity:

- for Web Server Attacks, the anomaly status is linked to the target server;
- for Path Attacks, the anomaly status is linked to the target path;

- for Parameter Attacks, the anomaly status is linked to the tested parameter;

When the Hacking Task is performed through an Attack Integration, an *Anomaly Checker Module* needs to be implemented. This is usually performed by parsing the attack tool output or by analyzing the output of HTTP Responses.

3.2.3 HTML Analyzer

The platform needs to know how to interact with the target. The communication with the target is basically based on HTTP Requests. Sending HTTP requests that contain valid parameter values is crucial as the application could generate errors and compromise the attacks. *HTML Analyzer* module is used to parse HTML pages and generate valid HTTP requests. [27] describes a method to automatically fill in web forms in HTTP requests. We use a variant of this approach that leverages Natural Language Processing techniques to identify semantic similarity between textual information extracted from the attributes of UI components and predefined data types (string, number, fiscal code, etc.). In this way, it is possible to create HTTP requests that the target web server does not reject. The generated HTTP requests are used to fulfil web attacks. The inner details of the model are beyond the scope of the current work.

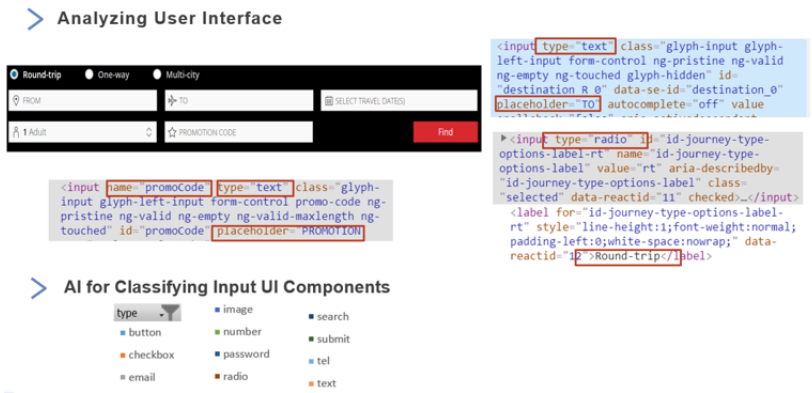

Figure 3.4: Html Analyzer Module

Listing 5 shows the input schema.

```
{
    "id_req": integer,
    "url": string,
    "header": [
        {string: string},
        ...
    ],
    "body": string
}
```

Listing 5: Anomaly Checker Response

Key	Description
key	An integer identifying the current request an integer identifying the current request
id_req	The URL under analysis
url	Tthe HTTP headers received while retrieving the object, it is a list of dictionaries in the form name: value.
body	HTML content to be analyzed

Table 3.3: HMTL Analyzer input schema parameters

3.3 Implementation

In this section, we describe the implementation of the architecture devised in Section 3.1.

3.3.1 Executor

The executor component is a web server developed in Django, a Python-based open-source web framework. It uses a database to store the catalogue of implemented Hacking Tasks and the Attacks that compose them. In particular, PostgreSQL has been used for this implementation. An object-relational mapping layer (ORM) is used to support the interaction with application data, regardless of the chosen relational database implementation. Users leverage the Executor's functionality through a frontend application, which acts as a client towards the executor module. Frontend and Executor together adhere to the REST architecture style. The Executor exposes RESTful APIs, which are used

for two main purposes:

- CRUD functions for persistent storage on the database;
- imparts the execution of HTs and single attacks to the Executor.

In Table 3.4 are described some of RESTful APIs that allow running an HT or a single attack. Choosing to run an HT implies that the platform autonomously decides which attacks are necessary to complete the current objective. However, users can also choose to run a single Attack.

Name	Method	Description
`/run-ht/<id_project>/ <id_hacking_task>/`	POST	execute the HT against the target identified by id_project.
`/projects/<id_project>/ attack/<id_attack>/`	POST	run an attack of category "web-server" identified by id_attack against the base URL of the web server identified by the id_project.
`/paths/<id_path>/ attack/<id_attack>/`	POST	execute an attack of category "path" with id equal to id_attack towards the Path identified by id_path.
`/params/<id_param>/ attack/<id_attack>/`	POST	execute an attack of category "parameter" with id equal to id_attack using the parameter identified by id_param.

Table 3.4: Executor Operations

Users create a project, which identifies a penetration testing session towards a specific target, by recording its base URL. A project is assigned an ID.

The Executor takes charge of a request for running an Attack or an HT through the RESTful API. Upon finishing the execution, the Front-end is notified asynchronously through an open-source message broker called RabbitMQ. The Streaming Text Oriented Messaging Protocol (STOMP) was used as a communication protocol, adopting a Publish/Subscribe interaction pattern. The Executor works as the Publisher, while the Front-end represents the Subscriber.

The message broker is an essential component of the implemented architecture, as it also serves other purposes. In fact, since attacks and tasks are time consuming activities, the Executor has been implemented in order to provide support for long running jobs, such as:

- **run_hacking_task**;
- **run_attack**;
- **mitm_proxy_receiver**;

In particular, Celery was selected to provide an asynchronous task queue based on distributed message passing. Thanks to one or more workers, it can receive tasks through a message broker and run them concurrently.
The Executor uses an HTTP proxy to intercept requests and responses made during an HT or an attack. This component sends the requests and responses intercepted to a Celery task, called mitm_proxy_receiver. Figure 3.5 shows how the MITM Proxy intervenes in the execution of an HT. The Attack executed is called "dirb", an open-source web content scanner in the picture.

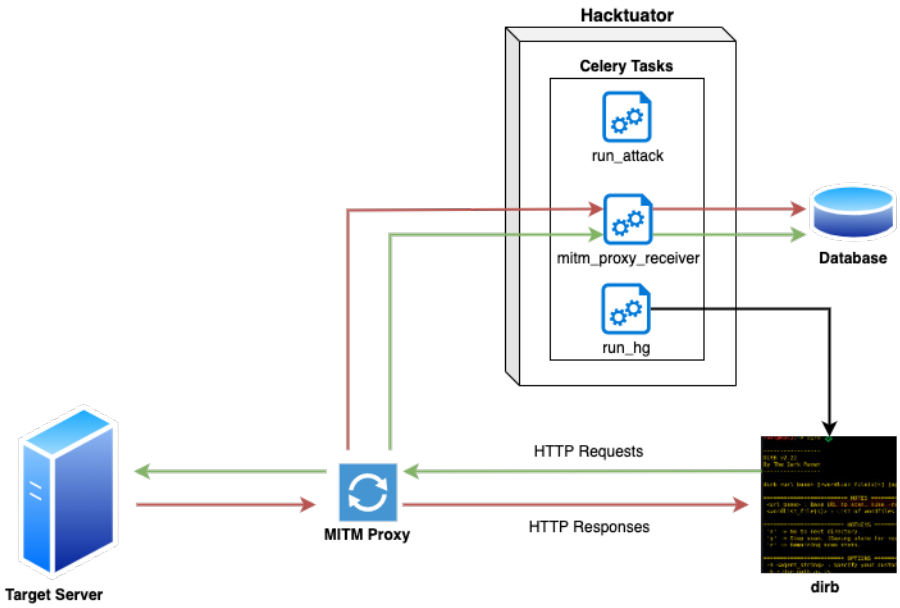

Figure 3.5: MITM Proxy component.

1. The Frontend sends the Executor a request to perform an HT.

2. Celery creates a new asynchronous task, run_ht, which performs an attack that consists of sending multiple requests to the target;

3. Requests performed during the attack, as well as their responses, are collected by MITM Proxy;

4. MITM Proxy sends the collected requests and responses to the Celery mitm_proxy_receiver task;

5. mitm_proxy_receiver stores the requests and responses within the database. Requests and responses are later analyzed by the Orchestrator module to look for anomalies or vulnerabilities.

The communication between MITM Proxy and the mitm_proxy_receiver task takes place via the RabbitMQ message broker.

3.3.2 Orchestrator

The Executor needs to interact with the Orchestrator to orchestrate its actions and update the environment's information.

Endpoint	Method	Description
html-analyzer	**POST**	The Executor sends html pages to the Orchestrator to identify the entry points in the pages.
anomaly-checker/	**POST**	This endpoint is contacted from the Executor to check for anomalies.
action-suggester/	**POST**	This endpoint is contacted from the Executor to ask the Orchestrator for actions to be performed in order to complete an HG or attack.

Table 3.5: Orchestrator REST endpoints

To provide these interactions, the Orchestrator exposes RESTful APIs. Such endpoints are used by the Executor, which acts as a client towards the Orchestrator module. Table 3.5 outlines the REST endpoints exposed by the Orchestrator. The endpoints enable the communication with "Anomaly Checker", "HTML Analyzer", and " Suggest Next Attack"

3.4 Attacks Integration

In this Section, we show how it is possible to integrate several types of attacks into our system.

It is possible to categorize several types of attacks and actions:

1. ***HTTP Request Attack***: A simple attack is to just send an HTTP request against the target and obtain an observation. An Action contains an Observable that analyses HTTP Response to verify a vulnerability or an anomaly. These attacks are helpful when you build a custom attack composed of several actions;

2. ***Command-Based attacks***: these attacks are executed through a binary executable on a terminal, for example, dirhunt is a binary used to perform spidering of web applications;

3. ***API-Based attacks***: these attacks integrate tools that expose an API. Several Security Scanners, such as OWASP ZAP, offer such a functionality.

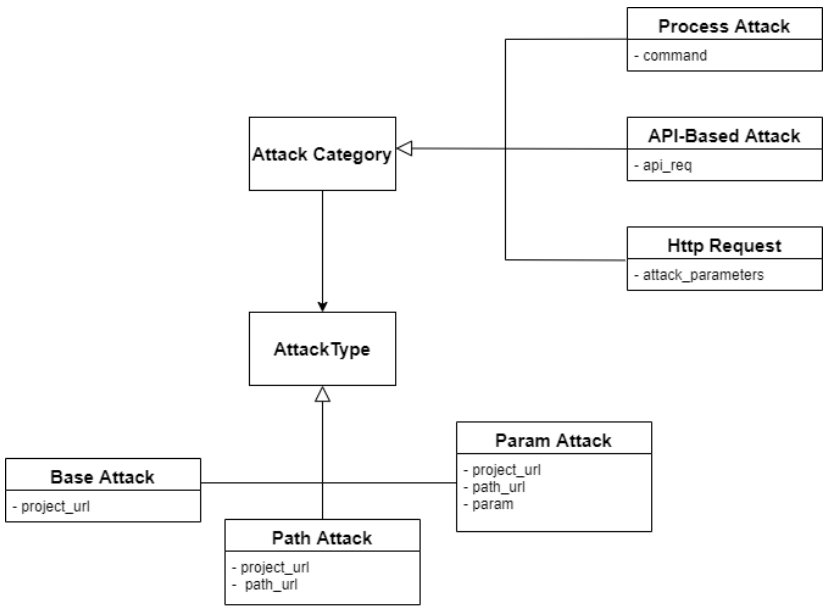

Figure 3.6: Attack Domain Classes

As described in Section 3.1, attacks can also belong to WebServer, Path or Parameter category. Figure 3.6 shows a class model for the attacks. We show how it is possible to integrate different tools by extending the attack class described previously.

```
class HttpSender:
 reqMethods = {
     "GET": requests.get,
     "POST": requests.post,
     "PUT": requests.put,
     "DELETE": requests.delete
 }
def send(self, r):
 """Send the http request """
 req_function = reqMethods[r.method]
 resp = req_function(r.url + ...))
 obj_headers = HttpPart.to_obj(resp.headers)
 return HttpInteraction(r, ..))

def send_multiple(self, reqs):
 responses = []
 """ Send multiple requests and take """
 self.send(reqs[0])
 return responses

def fuzz(self, r, p, vals):
 fuzzer = FuzzerGen(r, vals)
 fuzzer.choose_param(p)
 reqs = fuzzer.gen_reqs()
 return self.send_multiple(reqs)

 class FuzzAttack(ParamAttack):
  def __init__(self, name, data ...):
   super().__init__(name, data, analyzer, notifier)
   if ("values") not in data:
     raise AttackNotParamException("values")
    self.values = data["values"]
  def send_attack(self):
   # Send attack and return a json
   responses = self.http_sender.fuzz(
   self.base_request,
    self.chosen_param["name"], self.values)
   return responses
```

Listing 6: Fuzz Attack

3.4.1 Fuzzing Attack

We can define a *Fuzzing Attack* as an extension of a *Parameter* Attack: it takes a list of N payloads and sends N HTTP Requests against the target. *Fuzzing Attack* (Listing 6) extends the *ParamAttack* class and uses the *HttpSender* utility class to send a list of requests against the target. The *Fuzzing Attack* class can be considered as the base class for implementing any injection attack .

3.4.2 Dirhunt attack

dirhunt is a spidering tool that allows the attacker to discover all urls inside a web application. It is possible to integrate *dirhunt* in our system by extending the Process Attack class.

```
class DirHuntAttack(ProcessAttack):
     init(self, name, data, analyzer):
          # Use Dirhunt attack as WEBSERVER TOOL
          # data["path"] = "/"
          super().__init__(name, data, analyzer)
          self.command = "dirhunt"
     def send_attack():
          return super().send_attack()
```

Listing 7: Dirhunt Attack

When the attack is performed, the HTTP Responses are intercepted by the *Mitm Proxy* Module and stored in the internal Database. Those HTTP requests could be used for further attacks.

3.4.3 Zed Attack Proxy

Zed Attack Proxy is one of the world's most popular free security tools. It is a Dynamic Security Scan that can be used to automate Security Assessment. It offers an API that allows to send scans and integrate the tool into other systems. As API Authentication, ZAP uses an HTTP header (*X-ZAP-API-KEY*) that should be sent during the scan request. It is possible to integrate *ZAP* in our platform by extending the API-Based Attack class.

```
class ZapAttack(ApiBasedAttack):
   init(self, name, data, analyzer):
   self.api_req = '<ZAP_URL>'
   self.headers['X-ZAP-API-Key'] =
      config.get('zap-api-key')
  def send_attack():
   return super().send_attack()
```

Listing 8: ZAP Attack

3.5 Behavioural Model Integration

This Section shows how it is possible to implement a Behavioural Model of a Hacking Goal inside our solution. As Proof Of Concept we choose the following Hacking Goal:

Detect Reflected cross site scripting

The Goal is to detect Reflected cross-site scripting vulnerabilities inside a Web Page of the target system. In cross-site Scripting vulnerabilities, malicious JavaScript Code is sent as part of the user's input, and the input itself is reflected inside the response Web Page. The victim's browser executes JavaScript code. The attacker that sends a malicious payload can obtain the victim's session cookie, deface the website, control user actions, etc. A simple flow to detect Reflected cross-site scripting vulnerability is the following:

1. Identify all the input parameters and submit a generic string that does not appear anywhere within the application;
2. Verify that the generic string is reflected;
3. Detect the reflection context;
4. Depending on the reflection context, send different types of payloads.
5. If the vulnerable website applies filters to the received input, try to bypass them through smart encoding techniques.

3.5.1 Hacking Goal requirements

The user should choose an HTTP request with input parameters to start the Reflected cross-site Scripting hacking goal. They should also choose the parameter that is going to be tested. Finally, the Executor should be able to send

a valid HTTP request. These requirements could be satisfied by other Hacking Goals. For example *Spidering the Web Application*, that traverses all hyperlinks in the Web Application Target and *Detect valid HTTP Request* that uses the *HTML Analyzer* module to parse the HTML page, find valid parameters and generate a valid HTTP Request.

3.5.2 Behavioural and Attack Model

We do not explain the used behavioural model in detail as the focus of this work is to show how it is possible to integrate pre-existing models inside our architecture. To the purpose, we leverage a simplified version of our work about using Reinforcement Learning to discover reflected cross-site scripting vulnerabilities [28]. The rationale is that an attacker starts by sending a simple request containing a generic string to detect a reflection. Then, he/she tries to send attacks and bypass filters depending on which input is reflected and which filters the server applies. As it is possible to observe, it is a sequence of attacks, depending on the acquired information.

It is possible to model a single attack with a *PayloadManipulation* attack. *PayloadManipulationAttack* is an extension of a *FuzzAttack*: it accepts a list of rules and generates a fuzzing list used to fuzz the parameter under test.

Some of the relevant rules in the model are:

- ***Payload type***: the base payload to use for the manipulation;
- ***Double Quote Encoding***: which encoding to apply on double quote characters;
- ***Quote Encoding***: which encoding to apply on single quote characters;
- ***Tag Encoding***: which encoding to apply on bracket tags;
- ***AppendToPayload***: which characters to append to the base payload;
- ***PrependToPayload***: which characters to prepend to the base payload;

Some of the relevant observations in the model are:

- ***Reflect Location***: where the string is reflected (inside the HTML code, as an attribute of an HTML tag, etc.). It can influence the choice of the attack payload;

- ***Reflection Status***: whether the string is perfectly reflected, encoded or filtered;
- ***Single Quote Defense***: whether quotes are filtered, encoded, escaped, or left unchanged;
- ***Double Quote Defense***: whether double quotes are filtered, encoded, escaped or left unchanged;
- ***Bracket Tags Defense***: whether bracket tags are filtered, encoded, escaped or left unchanged.

Figure 3.7 shows the details of the messages exchanged between Executor and Orchestrator during the execution of the Goal.

next_attack field, suggested by the Orchestrator, is the *PayloadManipulation* attack. *attack_properties* field inside *next_attack* message contains the "manipulation" vector, i.e., the instructions on how to manipulate the value of the parameter under test. In this simple flow example, the server does not apply encoding or other defences, and the input is reflected inside the HTML body.

1. The Executor asks for the first action when the Goal is *Detect reflected cross-site scripting*;
2. The Orchestrator suggests to send a Generic String as the first payload;
3. The Executor sends the Generic String against the target, obtains information about the reflection location and asks the Executor for the next attack;
4. The Orchestrator, according to its internal behavioural model, knows that if a string is reflected in HTML body, the best action to perform is to send a script payload attack;
5. The Executor sends the attack and returns the observations back to the Executor;
6. The Orchestrator knows that when the attacker sends a script payload, and a perfect reflection occurs, it means that a cross-site scripting vulnerability is present. Hence, it notifies the Executor to stop.

At the end of this flow, the Executor can ask if the parameter is vulnerable to cross-site scripting and the Orchestrator acknowledges this.

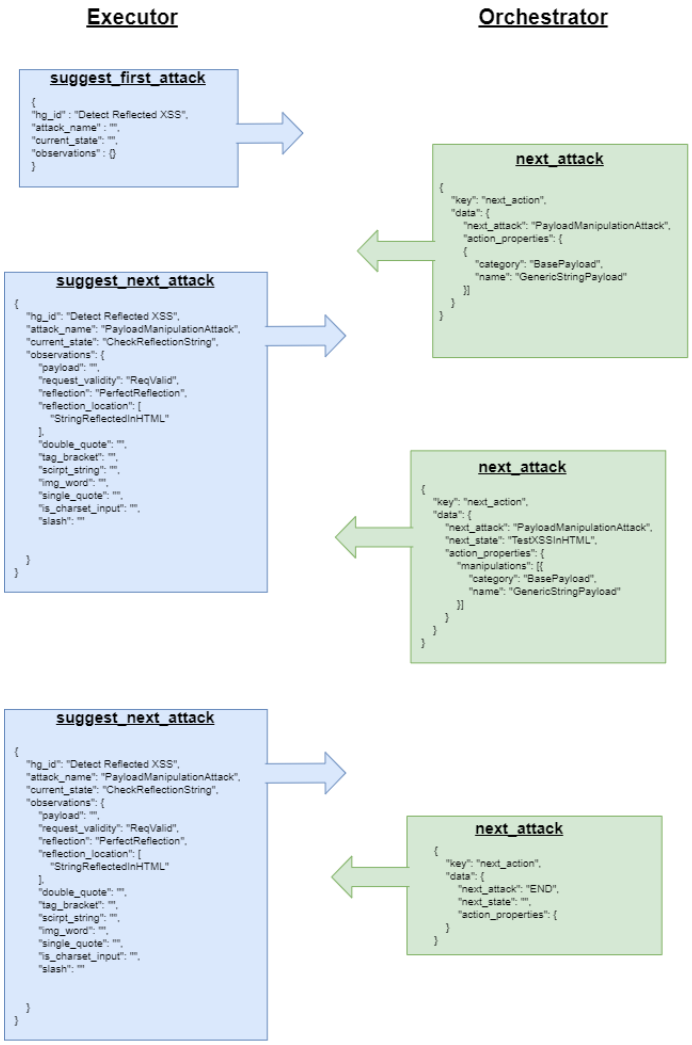

Figure 3.7: Executor Orchestrator flow

3.6 Vulnerabilities, Attack Tools and Behaviours

Several works [29] [30] have committed to the Web vulnerabilities classification. [31] describes 41 different types of Web vulnerabilities. We map each

vulnerability onto an attack or behavioural integration to evaluate the coverage of vulnerabilities through our framework. Table 3.6 shows the mapping between vulnerabilities, attack tools and behavioural models. Twelve vulnerabilities are covered by integrating an attack, thirteen vulnerabilities are covered by using a behavioural model and two vulnerabilities are covered by a combination of Attack Tools and Behavioural models. Fourteen vulnerabilities cannot be covered by our framework:

- **Cipher Transformation Insecure**: these vulnerabilities can be detected during the secure design phase;
- **Credential Management**: this vulnerability can be detected during the secure design phase, as well as the secure code review phase;
- **Insecure cryptographic storage**: this vulnerability cannot be detected without source code review. Hence, it can be detected during the secure code review phase;
- **Insecure deserialization**: this vulnerability is pretty difficult to find without looking at source code. It can be detected through secure code review activities;
- **Insecure digest** : this vulnerability can be detected during the secure design and secure code review phases;
- **Insufficient logging and monitoring**: this vulnerability can be discovered by reviewing the source code and by analyzing internal server logs. It is not possible to find it by using Web Application Penetration Testing techniques;
- **Injection Flaw**: this vulnerability is too much generic. It is indeed covered by other types of vulnerabilities, such as Cross-Site Scripting and SQL Injection;
- **Missing PT_DENY_ATTACH** is related with iOS mobile applications and it is not covered by our system;
- **Race Condition**: the discovery of this vulnerability is difficult. It can be detected by using approaches such as Manual Source Code Review, or *concolic* testing [32];
- **Security Misconfiguration**: this vulnerability should be reviewed in Web Server Hardening processes [33];

- **Sensitive data exposure**: it is too generic a vulnerability, it should be analyzed in several security phases;
- **Remote Code Execution**: this kind of vulnerability is more related to the exploitation phase. Actually, our platform is focused on vulnerability detection rather than exploitation;
- **Unvalidated automatic library activation**: this vulnerability is more related to third-party library management processes.

Vulnerability	Integration
Broken access control	B
Broken authentication	B
Carriage Return and Line Feed (CRLF) Injection	A
Cipher transformation insecure	O
Components with known vulnerabilities	A
Cross-Origin Resource Sharing (CORS) Policy	A
Credentials management	O
Cross-site request forgery (CSRF)	B
Cross-site scripting (XSS)	A/B
Directory indexing	B
Directory traversal	A
Encapsulation	B
Error handling	B
Failure to restrict URL access	B
HTTP response splitting	A
HTTP verb tampering	A
Improper certificate validation	A
Injection flaw	O
Insecure cryptographic storage	O
Insecure deserialization	O
Insecure digest	O
Insecure direct object references (IDOR)	B
Insufficient logging and monitoring	O
Insufficient session expiration	B
Insufficient transport layer protection	A
Lightweight Directory Access Protocol (LDAP) injection	A
Malicious code	O
Missing function level access control	B

Missing PT_DENY_ATTACH	O
Operating System (OS) command injection	A
Race condition	O
Remote code execution (RCE)	A
Remote file inclusion (RFI)	A
Security misconfiguration	O
Sensitive data exposure	O
Session ID leakage	B
SQL Injection	A/B
Unrestricted File Upload	B
Unvalidated automatic library activation	O
Unvalidated redirects and forwards	A/B
XML External Entities (XXE)	A

Table 3.6: Vulnerability and Integration Mapping

A = Attack tool Integration, B = Behavioural Integration, B/A = Behavioural and Attack Tool Integration, O = Other

Table 3.7 maps the vulnerabilities onto the tools that are integrated in the platform.

We cover ten out of fourteen vulnerabilities by using Open-Source tools. Four vulnerabilities are discovered by implementing custom tools:

- *Insufficient transport layer* protection is detected by analyzing all target endpoints and detecting if there is some unsecure plain-text communication. This can be performed by inspecting the used protocols (i.e., HTTP instead of HTTPs);
- *HTTP Response Splitting* is implemented by sending several HTTP requests and appending CRLF payloads as described in [44];
- *LDAP, OS and XXE Injection attacks* are implemented by extending the Fuzz Attack class through common fuzz strings.

We implement Attack Behavioural Models by using Port Swigger [45] and OWASP Testing Guide methodologies [46], as well as by implementing them through the Communication Protocol.

Vulnerability	**Attack Tool**
Carriage Return and Line Feed (CRLF) Injection	CRLF-Injection-Scanner [34]
Components with known vulnerabilities	Vulners API [35]
Cross-Origin Resource Sharing (CORS) Policy	CORScanner [36]
Cross-site scripting (XSS)	XSSMap [37]
Directory traversal	LFISuite [38]
HTTP response splitting	Custom
HTTP verb tampering	nmap http-methods script [39]
Improper certificate validation	MassBleed [40]
Insufficient transport layer protection	Custom
Lightweight Directory Access Protocol (LDAP) injection	Custom
Operating System (OS) command injection	commix [41]
Remote file inclusion (RFI)	fimap [42]
SQL Injection	SQLmap [43]
XML External Entities (XXE)	Custom

Table 3.7: Vulnerability and Attack Tool Mapping.

Chapter 4

Vulnerable Environments for Research and Education

To test the automation system described in the previous chapter, it is essential to have a heterogeneous vulnerable environment, that should be virtualized to avoid disruptions and data loss in real systems. In early research works, we designed a microservices-based framework to train students in Network Security, called Docker Security Playground [47]. The use of a microservices-based technology led to several advantages in terms of scalability, performance and usability [48]. During our exploration, we also ran into the limitations of this kind of virtualization when applied to the cyber-range domain. In order to overcome them, we implemented a hybrid system relying on different types of virtualization. A further contribution has concerned the definition of a model for cyber ranges, called "Cyber Range Environment" and the design of a secure architecture for cyber ranges deployment in cloud environments. We show how it is possible to integrate the Docker Security Playground in this model by provisioning it in a cloud enviornment. We use the Amazon Web Services cloud as a proof of concept to show our results.

4.1 A heterogeneous environment for cyber-ranges

IT security experts usually face problems that require across-the-board skills in order to design secure infrastructures, test them to avoid vulnerabilities and harden them against potential attacks. On the other hand, researchers try to apply ground-breaking solutions to well-known security issues. Indeed, a thorough research process is always accompanied by a good educational activ-

ity and vice-versa. Both research and training tasks require controlled environments inspired by real-world operational facilities that work as spaces to experiment new solutions as well as to practice technical skills. A "Cyber Range" is a virtual environment for cybersecurity testing, aimed at assessing defense methodologies and infrastructures countering modern threats. It is explicitly designed for training purposes. Everything that is needed for carrying out the activities and completing the course is inside the architecture: virtual machines, networks, tools, etc. Moreover, the environment must be well controlled in order to provide secure access to students who take part to the training, as well as administrators looking after management tasks. To date, there is no unique way to prepare cyber ranges and few solutions have been proposed so far, to the point that most of them are created manually. This matter raises several challenges, from the preparation of the vulnerable environment, to the automation of the initial configuration. In this regard, modern virtualization techniques play a fundamental role in the realization of effective cyber ranges. In particular, container-based virtualization offers different benefits in terms of scalability, performance and usability Though, this kind of virtualization is not able to reproduce all types of vulnerabilities, as we will point out in Section 4.1.1. To tackle this issue, we propose the design and implementation of a hybrid solution for the realization of a fully-fledged cybersecurity playground. The goal of our work is to provide a solution combining different virtualization techniques and paradigms to create a cyber-range environment that allows to take advantage of container-based virtualization without losing anything in terms of cybersecurity vulnerabilities coverage. In the proposed solution, networks, virtual hosts and containers linked to different technologies seamlessly co-exist.

4.1.1 Design

OS Virtualization and Vulnerabilities

In our work, we consider OS virtualization as a means to emulate vulnerable environments that can be attacked during training exercises.
There exist different types of vulnerabilities, as well as several ways to categorize them from the adversarial point of view. In order to combine the need for the trainees to learn common attack models and to familiarize with novel vulnerabilities, an effective way to design a cybersecurity exercise is in the form of a Capture The Flag (CTF) environment. In this sort of scenarios, users need to take advantage of vulnerabilities that allow them to get a first access to

a remote system. Such vulnerabilities concern applications or remote services like web servers and depend on either buggy implementations or misconfigurations.
Once gained access to a remote machine, users look for vulnerabilities that allow to acquire special privileges. Such vulnerabilities can depend on implementation and misconfiguration as well, but in this case they can occur both in user and kernel space.
Gaining special privileges is representative of the fact that attackers are in complete control of the system and can perform several other harmful operations, such as data exfiltration and lateral movement. The latter, though, depends on both the vulnerability of the machine and the configuration of the network infrastructure.
The choice of introducing OS virtualization, automatically rules out the emulation of kernel space vulnerabilities. On the other hand, designing vulnerable machines as microservices offers several advantages in terms of:

- *decoupling*: application dependencies can be installed and managed separately for each microservice;
- *scalability*: a single host can handle up to hundreds of containers [48];
- *provisioning*: container resources can be allocated taking into account the requirements of the implemented services, as well as the scalability needs.

The mentioned attributes are of course helpful during deployment. However, they become very profitable during design and testing as well. In fact, the activity that is proven to be the most resource consuming in the life cycle of a cybersecurity exercise, is the *preparation* [49] of the vulnerable environments, because it deals with the configuration and automation of heterogeneous systems. The designers of the scenarios can benefit from a technology that allows them to deploy the implemented scenarios in lightweight testing environments.

Hierarchical architecture overview

Isolation among emulated scenarios is the first fundamental requirement to ensure: this allows separate teams to perform training in a dedicated environment. With the term "**Virtual Scenario**", we identify the component that allows to implement and deploy the scenario. It is made up by a type-2 hypervisor and its guest virtual machines. The "Virtual Scenario" lands on what

we call a "**Master Host**": it represents a bare-metal hypervisor that allows to replicate the Virtual Scenarios, according to the amount of teams that are going to perform the training.
The choice of separating the two components allows the Scenario Designer to create the exercises on a testing environment with much more limited resources than the actual environment used for the final deployment. In fact, type-2 hypervisors are common tools supported by any desktop operating system and can be found on modern laptops.
The guest virtual machines inside the Virtual Scenario assume separate roles, each of them justified by the need of having:

- an *entry-point* to provide each team with remote access to the emulated scenario;
- one or more *container hosts* allowing to deploy the sections of the scenario that use OS virtualization;
- one or more generic *guest virtual machines* allowing to reproduce the sections of the scenario that can not use OS virtualization.

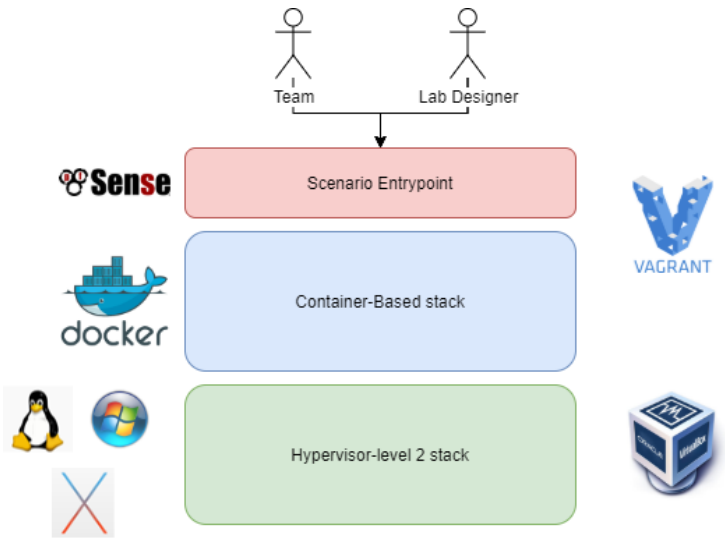

Figure 4.1: Virtual Scenario Architecture

Networking configuration

The integration among separate virtualization technologies also raises challenges for the network communications among components. In particular, the following issues need to be addressed:

- users that get remote access must be forwarded inside the network of the scenario, in order to start the training exercises;
- containers need to be able to communicate with other containers as well as other guest virtual machines. This allows the designer to have complete freedom over the network configuration of the scenario;
- basic routing configurations must be available to the designer, in order to separate those network segments with different semantics (e.g., an exercise with both a public and a private network segment).

To fulfill the first requirement, we decided to identify the entry point of the scenario with a Virtual Private Network server. In this way, upon connection, users are provided access to a network and can start building an understanding of the infrastructure. As for the second issue, we first make sure all guest virtual machines are part of the same VPN. Then, we want segments of the network to be populated with both containers and virtual machines. For instance, we would like some services to be handled by containers in one segment of the network and only virtual machines in another. These design choices are made in order to provide the scenario designers with enough flexibility. To tackle this issue, we introduce the concept of "**container as a router**". The expression refers, in fact, to containers with multiple network interfaces:

- one or more interfaces are attached to container networks;
- one interface works as a bridge towards one of the physical network interfaces of the guest virtual machine upon which the containers are deployed.

With the proper routing configurations, a network composed of sole containers can exchange packets with virtual machines. This allows to fulfill also an "information hiding" design principle: users that solve the exercise just see network services, regardless of their implementation in the form of either a container or a virtual machine.
Proper routing configurations allow to solve the last of the three requirements

as well. In the implementation section we will underline how these concepts are put into practice thanks to the introduction of known technologies in the field of OS virtualization and by leveraging the Infrastructure as Code paradigm.

4.1.2 Implementation

In this section we will describe the implementation of the architecture devised in section 4.1.1.

OS Virtualization. For the implementation of the container-based segment of the virtual scenario, our choice fell on Docker. With Docker, we can design vulnerable Linux services using the Dockerfile syntax. On the other hand, we can build networks made up by multiple containers by using docker-compose, a tool providing orchestration functionality for multi-container applications.

As a popular OS virtualization technology, Docker inherits both the advantages and the limitations identified in section 4.1.1. Although it is possible to design Windows services as Docker containers thanks to the native support introduced since Windows Server 2016, we decided to use Docker containers only for Linux-based services, since the Docker for Linux community provides a much richer support. The Windows-based vulnerable services will be implemented in standard virtual machines.

In table 4.3, we identified the common categories of vulnerabilities used in training exercises: for each of them, we reported whether they can be reproduced using Docker or not, according to the considerations made in section 4.1.1. This taxonomy will allow us to keep track of the actual vulnerabilities covered in the final evaluation, provided in section 4.1.3.

Container as a router. In section 4.1.1, we introduced the concept of "container as a router" as a means that allows containers belonging to separate networks to exchange packets between each other. Such networks can be composed of either docker containers, or other guest virtual machines. The latter case is achieved using the so called "*macvlan*" driver, that creates a bridge between the network interface of a container and a provided physical network interface. In this way we allow a container to exchange packets with a network composed of guest virtual machines. Such solution is depicted in Figure 4.2.

Virtual Scenario Provisioner. The main advantage of many existing cyber ranges instantiation platforms is that both configuration and deployment of the scenarios are consistently automated. We provided a fair amount of automation as well, by embracing the Infrastructure as Code (IaC) paradigm,

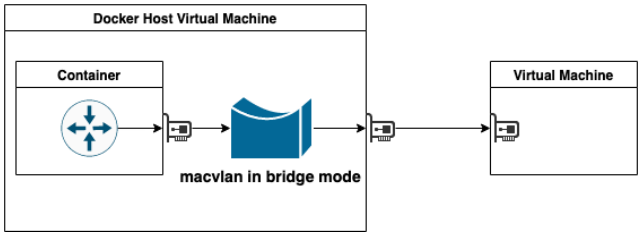

Figure 4.2: Container as a router

that allows provisioning complex systems through machine readable files. In particular, we want the scenario designers to be able to launch an architecture made up by several virtual machines with minimum effort. The Vagrant software allows to do so, thanks to the definition of a so-called Vagrantfile, that contains the description of a virtual environment, together with its network configuration. The Vagrantfile can be seen as the “Virtual Scenario file", as it defines the list of guest virtual machines that form the training exercise. It includes also information about the type-2 hypervisor of choice for the implementation of the exercise, as well as the definition of the network infrastructure.
The other aspect of the architecture that responds to the IaC principle, is of course the orchestration of Docker containers inside the Docker virtual host. Such orchestration is handled with a docker-compose file, that describes the services deployed in the form of docker containers and the subnets they belong to. It also allows to specify the networking driver used, which is important when the “container as a router” concept needs to be implemented. The two files mentioned above must be provided by the scenario designers to leverage the proposed architecture.

Other configurations. To properly configure a virtual scenario, the designer also needs to provide scripts that allow, at the startup, to fully prepare the vulnerable environment:

- **Entrypoint Configuration Scripts** to properly configure the scenario entrypoint. For that, we rely on a PfSense virtual machine, which works as a VPN server, whose initial configuration allows users to be directly connected to the vulnerable network infrastructure of the training exercise. The configuration is reusable by the designers;

- **Network Configuration Scripts** to allow containers belonging to separate docker subnets to communicate with each other, as well as to com-

municate with other virtual machines through the macvlan driver. In the proposed architecture, the "container as a router" is a Linux-based docker container. Its configuration is made with Linux common network utilities such as *ip*, *route* and *iptables*. Such scripts are only partially reusable by the designer, as they depend on the specific network infrastructure required to reproduce the training exercise;

- **Vulnerability Scripts** to configure the vulnerable services. Such scripts need to be customized by the designer as they depend on the training exercise being implemented. The configuration of the vulnerable services is outside of the scope of our work.

4.1.3 Evaluation

In this Section we first show the variety of vulnerabilities that we are able to reproduce using the proposed solution, by describing an ad-hoc emulated scenario. Then, we discuss the benefits that derive from the usage of OS virtualization.

Emulated scenario. To prove that our approach benefits from the introduction of containers, whilst making sure that this choice does not affect the realism required for a training exercise, we need to:

- implement at least one vulnerability per each category of vulnerabilities identified in Table 4.3;

- create a configuration that closely mimics a real-world network infrastructure as well as enables communication among heterogeneous virtualized environments.

In table 4.1 we summarize the elements that compose the virtual scenario we implemented. Hypervisors are listed on the left side, while on the right side are reported the guest virtual machines/containers that have been chosen to implement the vulnerabilities. We use a PfSense guest virtual machine to work as Scenario Entrypoint. Then, we have an Ubuntu 18.04 LTS guest virtual machine that works as Docker Host. It runs 10 Docker containers: 8 of them are service containers used to reproduce vulnerable machines; the remaining 2 play the role of "router containers" and are part of the network infrastructure. Then, we have the section of the scenario dealing with vulnerabilities not reproducible using Docker containers. Namely, Windows machines and an Ubuntu guest VM equipped with a vulnerable Linux Kernel.

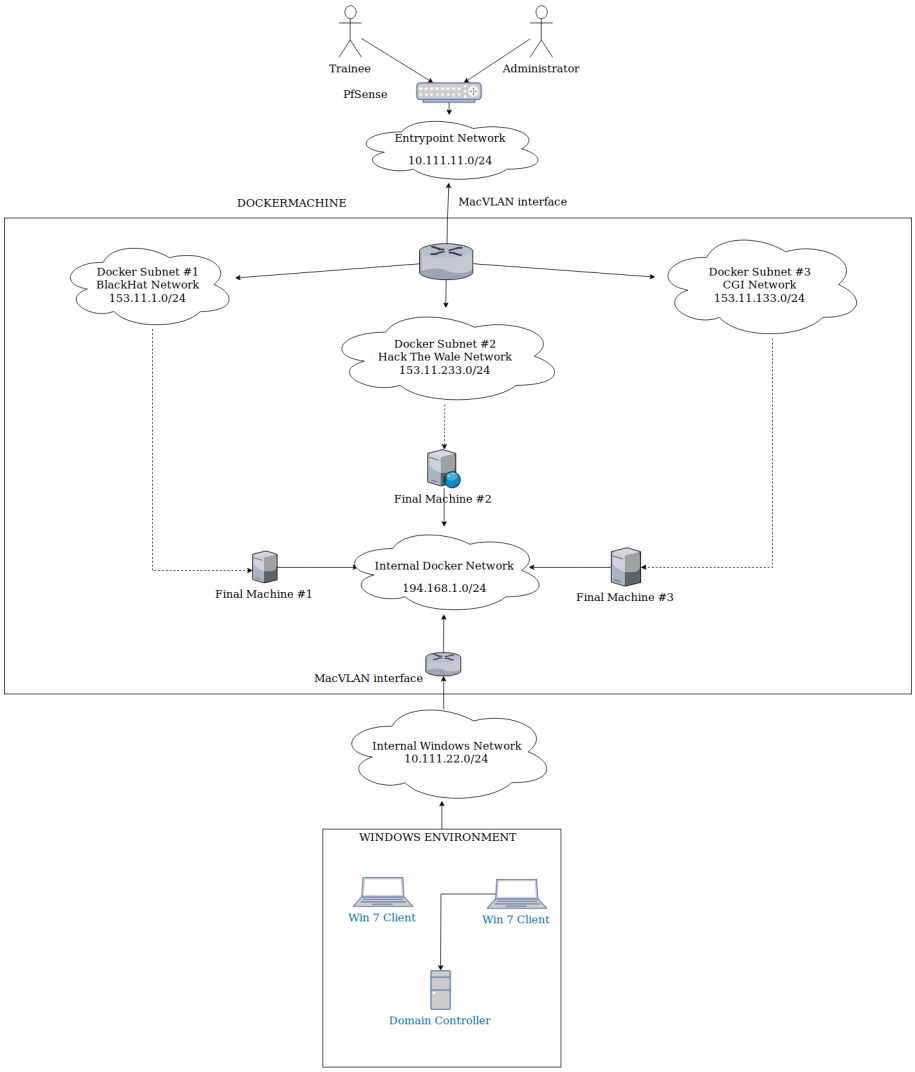

Figure 4.3: Virtual Environment Architecture

Table 4.2 provides a complete list of the vulnerabilities covered in the scenario and maps them onto the categories listed in Table 4.3. Using our solution, we ensure that the identified categories of vulnerabilities can be reproduced, by taking the benefits of the OS virtualization approach, while at the same time overcoming its core limitations.
As for the second goal of the evaluation, we decided to recreate a scenario with corporate network infrastructure, made up by: (i) a private segment, with internal hosts not accessible from unauthorized users; (ii) a *DeMilitarized Zone* (DMZ) to provide external users with access to public services.

The complete virtual scenario is showed in Figure 4.3. During the training, the purpose of the users is to get as deep as possible inside the network and find vulnerabilities allowing them to acquire unauthorized access to the private section of the network, a practice known as "*lateral movement*".
Such a network infrastructure allows us to show the flexibility of the proposed architecture, which provides many configurable options, by allowing the scenario designers to realize cybersecurity exercises with a high degree of realism.

Discussion. The virtual scenario was first designed and tested on a laptop with an Ubuntu 18.04 LTS (on a 64 bits processor) and 16GB of RAM. For final deployment, a Vmware ESXi platform was used to replicate the scenario for 10 separate teams. The architecture was leveraged as a training exercise for the Italian "Joint Cybernetic Operations Command"[1]. Each team, formed by expert penetration testers, was provided with a VPN certificate, in order to connect to separate virtual scenarios. All of them reported a smooth user experience and the difference between vulnerable containers and virtual machines was not spotted out by any of the participants.
We care to point out that, in order to implement the same scenario without OS virtualization, an amount of virtual machines three times larger is needed. In fact, using only type-2 hypervisors, each service container would be implemented as a guest virtual machine, resulting in a total amount of 12 guest virtual machines used for vulnerable services.
As future development, we plan to implement a Windows Docker Host, in order to implement Windows services vulnerabilities as Docker containers. This will allow us to remove the burden of the Windows virtual machines used in this scenario.

[1] Italian: Comando Interforze Operazioni Cibernetiche (CIOC)

VS Element	Implementation
Bare-Metal Hypervisor	Vmware ESXi Server
Hyper-V Scenario Entrypoint	PfSense Virtualbox Machine
Container-Based Stack Machine	Ubuntu 18.04 LTS Virtualbox Machine
Container-Based Stack Environment	10 vulnerable Docker containers 3 vulnerable networks, 1 public network, 1 internal network
Level-2 Hypervisor stack	1 Windows Server 2008 (Domain Controller), 2 Windows 7, 1 Ubuntu Xenial 16.04 64 bits

Table 4.1: Virtual Scenario Implementation Example Components

Type	Vulnerability
WAV	KikChat - LFI / RCE
MPEV	passwd file World-Writable
LAV	LFI - Local File Inclusion
SPEV	Mysql UDF running as root
LAV	Wordpress Vulnerability
LUPEV	vim with bit suid
LAV	Buffer overflow in custom application
MPEV	Same password for separate users
LAV	Shellshock on User-Agent header
LAV	Bruteforce attack against a vulnerable service
SPEV	Local Webserver with root privileges
LAV	CVE-2017-5645
LKV	Dirty-Cow vulnerability
NLAV	Windows FtpShell Client Buffer Overflow
NLPEV	Pass the Hash / SMB Relay via XSS

Table 4.2: Attack Scenario Vulnerabilities.

ID	Vulnerability Type	Can use Docker?
WAV	Web Application Vulnerabilities	Y
LAV	Linux-based Application Vulnerabilities	Y
SPEV	Privilege Escalation through services running with high privileges	Y
LUPEV	Linux-based User space privilege escalation	Y
MPEV	Privilege Escalation through Misconfiguration	Y
LMV	Linux Misconfiguration Vulnerabilities	Y
SV	Service Vulnerabilities	Y
NLAV	Non Linux-based Application Vulnerabilities	N
NLRV	Non Linux-based Remote Vulnerabilities	N
LKV	Linux Kernel-level Vulnerabilities	N
NLPEV	Non Linux-based Privilege Escalation Vulnerabilities	N

Table 4.3: Vulnerabilities and Docker applicability.

4.2 Deployment and orchestration of Cyber Ranges in the Cloud

Cyber ranges are evolving in more complex environments and simulations are getting closer and closer to real world scenarios. These innovations bring with them major technical challenges and complex problems to solve. To date, little has been done to automate the process of creation of cyber security training environments. Moreover, cyber ranges are recently evolving into service-based solutions. Remote Access Control, Credential Management, Automation and Security are the aspects that nowadays require innovation in this field and this is where our work focuses on. We propose a design of a services-based platform for the dynamic deployment of cyber ranges, based on containers and virtual machines. The goal is to manage the creation and termination of training environments in a dynamic, scalable and secure way. Moreover, the architecture satisfies the property of portability, which allows to consider its implementation with different cloud providers. In our case it was decided to use the cloud-computing platform offered by Amazon (Amazon Web Services — AWS), which is cheap, highly reliable and configurable. EC2 (*Elastic Computing Cloud*) instances are Virtual Machines configured with Amazon Machine Images (AMI), a type of virtual appliance that runs on top of a hypervisor. They play a central role in our work, as they are responsible for hosting Docker Security Playground (DSP) [47], a microservices-based framework whose architecture is based on docker[2] and docker-compose[3].

[2] `https://www.docker.com/`

[3] `https://docs.docker.com/compose/`

Currently, DSP can be used only by a single user. The proposed solution allows to implement DSP collaborative laboratories that involve interaction between *red* (i.e., the attackers) and *blue* (i.e., the defenders) teams through a shared environment. However, EC2 Instances do not only act as a mere "host" for DSP. They are also an active entity in the reproduction of cyber security scenarios, since they can run several different Operating Systems, hence allowing for the creation of complex and diverse virtual and hybrid environments. In such a scenario, environments separation and access control become key to the success of the cyber range. A user shall be able to access only resources that were assigned to him/her by the system. Moreover, these resources must be isolated from the outside, so to prevent internet access and protect the environment from known attacks to cloud infrastructures such as cryptojacking. Scenarios range among hacking web applications, cracking telnet access, buffer overflow, WiFi hacking, and many others. Users have access to a collection of multidisciplinary laboratories, which brings many benefits from an educational point of view. Much has been done also to automate creation and management of resources, ensuring a dynamic and reliable environment.

4.2.1 Cyber Range Environment

The Cyber Range Environment is a set of virtual machines that are logically divided into multiple subsets called, respectively, "Macro Ranges" and "Micro Ranges". A Macro Range *Mr* is a subset of the Cyber Range environment *C* that contains one and only one Remote Access Controller *r*. The Remote Access Controller is configured with a set of routing rules obliging users to gain access exclusively to the assigned virtual resources. These rules are defined inside the Remote Access Controller and cannot be modified by unauthorized entities.

$$Mr \subseteq C$$
$$\exists! \quad r \in Mr$$

A Macro Range contains zero or more Micro Ranges. Micro Ranges are a non empty set of virtual machines whose usage is predisposed for either a single user or a group of users.

$$n(mr) \in \{1, 2...M\}$$

Where M is a positive integer representing the maximum number of eligible Virtual Machines per Micro Range. All the Micro Ranges in a Macro Range must be linked to the Remote Access Controller through a communication channel. A virtual machine cannot establish in any way a communication channel with a virtual machine belonging to a different Micro Range.

A virtual machine *u* is *reachable* from a virtual machine *v* if a direct communication channel can be established between them.

$$v \rightarrow u$$

At least one element of a Micro Range must be *reachable* from the Remote Access Controller.

$$\forall \quad mr \subseteq Mr \quad \exists \quad v \in mr : r \rightarrow v$$

Elements of different Micro Ranges must not be able to reach each other.

The same rule applies to elements belonging to different Macro Ranges. Fig. 4.4 shows an example.

Let Q and S be two Micro Ranges with v and u virtual machines belonging to Q and S respectively.

$$\nexists \quad v \in Q \quad : v \rightarrow u \quad \forall v \in Q, \forall u \in S$$

$$\nexists \quad u \in S \quad : u \rightarrow v \quad \forall v \in Q, \forall u \in S$$

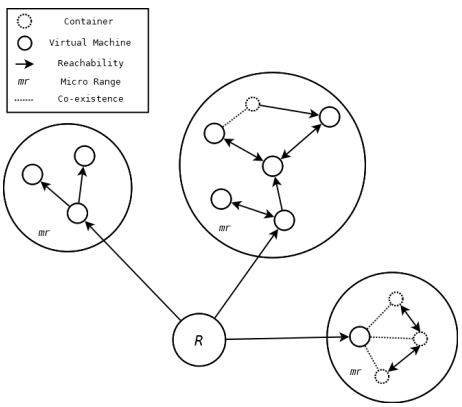

Figure 4.4: Representation of a Macro Range.

Micro Ranges are divided into multiple categories depending on the type of resources they are composed of:

- *Virtual Micro Range*: is a Micro Range composed only of full virtualization VMs connected to each other;
- *Containerized Micro Range*: is a Micro Range composed of a single VM whose kernel serves as the base for the execution of containers (Docker, Linux Containers, Solaris);
- *Hybrid Micro Range*: is a Micro Range composed of both full virtualization VMs and one or more VMs hosting containerized infrastructures. Container Networks (based, e.g., on *docker-compose*) and other VMs are connected to each other forming, de facto, a hybrid system;
- *Shared Micro Range*: this type of Micro Range can be accessed by several users at the same time. A shared Micro Range is designed to simulate red/blue team scenarios that allow interaction between attackers and defenders.

4.2.2 Back-end Resource Manager

The Back-end Resource Manager is responsible for:

- *Resource allocation*

  Users send resource allocation requests through the front-end application and the API. The Back-end Resource Manager receives such requests and checks if all the required preconditions are met, such as eligibility and accountability of the user who made the request. In case of laboratory creation, it checks if there are sufficient resources for allocating a new Micro Range in any Macro Range inside the Cyber Range Environment. If the maximum number of Micro Ranges per Macro Range has been reached it will proceed to create a new Macro Range, including a Remote Access Controller.

- *Routing rules declaration*

  When a new virtual machine is created, or when the ownership of a Micro Range passes to another user, the Back-end Resource Manager launches commands inside the Remote Access Controller to enable the new rules. A command is sent in order to establish a communication

channel between the user who made the request and the assigned Micro Range. In case of transfer of ownership to another user, the existing rules are modified. If the system needs to revoke a user's access to a Micro Range, the Back-end Resource Manager sends a delete command related to the rule that allows communication between such two entities.

The Back-End Resource Manager needs to be aware of the state of the Cyber Range Environment before setting up a communication channel between a user and a Micro Range. Before editing the virtual space, it reads the current state from the Data Storage and updates it every time a successful action on the Cyber Range Environment is performed. This state control mechanism prevents overlap of resource allocation or address assignment.

4.2.3 Cluster Security Controller

The Cluster Security Controller is responsible for interpreting events coming from the Event Listener and performing security checks and actions on the Cyber Range Environment. Security checks include verifying if there are anomalies in the allocation of the instances, for example if a user manages to create more resources than expected. The event record also includes information about the Micro Range owners. Let us suppose that a user is authorized to have access to only one Containerized Micro Range and, intentionally or due to a system bug, manages to get ownership of an additional Micro Range of the same type. The Cluster Security Controller is notified about this event and, after performing security checks, detects the anomaly and destroys the resources. The Back-end resource Manager tries to prevent this type of situations by performing controls before the resource allocation phase. Though, it does not check the environment in real time, differently from the Cluster Security Controller. The Cluster Security Controller, in fact, updates the state of the overall system by updating the Data Storage if the performed actions modify resources or communication channels in the Cyber Range Environment. The automated controls performed on the active resources in the Cyber Range Environment constitute an additional security layer which is crucial for architectures of such complexity.

4.2.4 Credential Manager

Accountability is one of the most important aspects in this type of systems. The Credential Manager is responsible for providing authentication and access

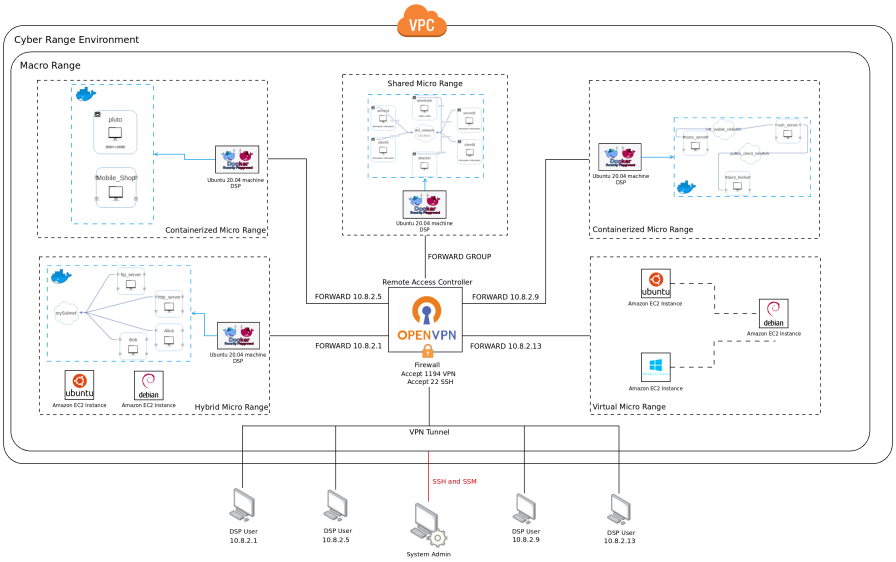

Figure 4.5: Cyber Range Environment implemented with AWS

control services to the users, who are in turn identified by unique attributes like username, email or phone number. Additional security attributes are required to set limits to the resources that can be requested and keep track of payments history.

4.3 Implementation

We show a possible implementation of the proposed architecture using Amazon Web Services. A potential scenario is shown in Fig. 4.5. There are four users, each with an assigned Micro Range. All four users are assigned to one Shared Micro Range that hosts a red/blue team laboratory. The Cyber Range Environment is implemented with a Virtual Private Cloud (VPC), i.e., a virtual network that allows to dynamically activate cloud resources in a controlled fashion. A Macro Range is implemented by allocating a private subnet inside the VPC. This subnet contains virtual machines, created as Elastic Compute Cloud (EC2) Instances. EC2 Instances are virtual machines created with a special type of virtual appliance called Amazon Machine Image (AMI). There are plenty of available Operating Systems in AWS, ranging from Linux distributions (such as Ubuntu, Debian, Kali, Fedora) to different versions of Windows

```
-A POSTROUTING -s 10.8.2.1 -o eth0
-d 172.31.89.138 -j MASQUERADE
```

Figure 4.6: NAT Rules

Server. The Remote Access Controller has been configured trough a Ubuntu 20.04 EC2 instance running an OpenVPN server. The Server is responsible for connecting users to the subnet and forwarding packets to correct destinations through the policies and NAT rules described in a firewall local to the instance. An example of NAT rule is shown in Fig. 4.6.

This rule means that all packets coming from the user whose virtual IP address is 10.8.2.1 will be forwarded to the assigned virtual machine, belonging to a Micro Range, whose private IP address is 172.31.89.138. POSTROUTING means that this rule must be applied on packets that are leaving the OpenVPN Server. In this way the user can access the services provided by the EC2 instance. The local firewall must leave ports 1194 and 22 opened, respectively to allow users to connect to the OpenVPN server and to allow administrators access to the server instance itself through a secure shell.

Containerized Micro Ranges are implemented with a Ubuntu 20.04 EC2 instance running the Docker Security Playground [47]. Some types of containers run vulnerable web applications that are accessible trough specific ports. Users have access to all available ports in their personal instance and in this way are able to launch hacking tools installed both in their local machine (the one used to connect to the OpenVPN Server) and local to the EC2 instance. In case of more complex scenarios requiring access to containers, the user can take control of the shell of the instance trough a secure SSH tunnel.

Granting shell access to users could expose the entire architecture to attacks and illicit actions. A user may perform network scanning and interfere with the correct functioning of Cyber Ranges assigned to other users. The use of these tools is only allowed inside the assigned Micro Range. AWS allows us to equip EC2 instances with a virtual firewall called *Security Group*, that ensures environment separation. A Security Group is set to accept or deny packets coming from specific private IP addresses of the subnet. In case of Containerized Micro Ranges the Security Group of the instance that hosts containers accepts incoming packets that have as source the private IP address of the Remote Access Controller (OpenVPN Server in our case), which acts as a relay for user traffic. In case of other types of Micro Range, this depends on

the type of virtual scenario that has to be simulated. One might want to allow incoming and outgoing traffic to a vulnerable EC2 instance and, after the user has taken control, allow him/her to reach another instance of the Micro Range. The other VM will have a Security Group set up to allow reachability only from the hacked EC2 instance.

The use case to implement is the following: a user sends a request for a trial Containerized Micro Range of 30 minutes duration and composed of a virtual machine running the Docker Security Playground. A user can launch only one trial Micro Range per account and after 30 minutes all the allocated resources are terminated by the system.

The role of the Credential Manager is performed by *Cognito*, an AWS service providing authentication, authorization and user management. When a user registers to the service, their personal data will be saved in a Cognito user pool. Access to the API for incoming requests is regulated by a *Lambda* authorizer that fulfills the task of validating JSON Web Tokens (JWT) in the request header and checking whether a user has already sent a request for a trial Docker Security Playground Micro Range or not. If the JWT token is correctly validated, the API adds the requester's username to the body field of the request and passes it to the back-end. If the user made the request for the first time, the Lambda Authorizer sets a Cognito attribute for that user as 'false', hence indicating that he/she cannot send another request for a trial instance.

The component Back-end Resource Manager is implemented in Python3 with a Lambda Function that performs the following operations:

- **Eligibility check**: before allocating resources the Lambda function checks if the request is valid and the user exists.

- **Availability check**: every Macro Range has a maximum number of users and Micro Ranges that it can handle. The Lambda function checks if a Macro Range exists that can host the new Docker Security Playground EC2 instance. If all Macro Ranges are occupied, the Lambda function generates a new one within the VPC (Cyber Range Environment), by creating a new subnet and a new OpenVPN server instance.

- **Virtual IP and ClientID generation**: once the OpenVPN server has been selected, the Lambda function assigns a virtual static IP address to the user. The assignment operation must take into account the already assigned virtual addresses. To avoid assignment overlaps, the Lambda function reads the current Cyber Range Environment state from the Data

Storage. Once the virtual static IP address has been randomly generated, it is marked as 'assigned' and the Cyber Range Environment state is updated.

- **EC2 instance allocation**: the Lambda function creates the EC2 instance using a custom Ubuntu 20.04 Amazon Machine Image with Docker Security Playground installed. The EC2 instance is tagged with an ad hoc value indicating that it is a trial instance and must be terminated after 30 minutes.

- **Commands execution on the OpenVPN Server**: the EC2 instance has a unique private IP address within the subnet that needs to be assigned to the virtual static IP address of the user with the routing rules previously discussed. The Lambda function sends configuration commands to the OpenVPN Server through the AWS System Manager, so to enable the iptables routing rule.

The Cluster Security Controller is implemented through AWS Step Function, a function orchestrator that allows for sequential execution of either Lambda functions or other AWS Services. The Step Function execution is triggered by AWS Cloudwatch, a cloud resources monitoring service acting as the Event Listener. When a new instance is allocated or changes its state (for example from Stopped to Running) within the VPC, Cloudwatch sends an event description in JSON format, which is processed by the Step Function. The Step Function invokes a Lambda Function that carries out security controls, including checking the tags of the instance. If the tag 'IsTrial' of the instance is set to 'true', the next step is to terminate the instance after 30 minutes, by invoking a Lambda Function that fulfills the purpose. The JSON event passed as input to the Lambda Function includes all tags associated with the instance to be terminated. After the DSP trial instance is terminated, the Step function updates the Cyber Range Environment state within the Data Storage by marking the virtual static IP address of the user as free, so that it can be assigned to another user. Moreover, the routing rules are deleted from the OpenVPN Server, once again by launching commands through the AWS System Manager.

Conclusions

This Thesis describes our research works aimed to automate Web Application Penetration Testing. The first research area that we explored was related to the definition of an attacker's behaviours. We define a goal-centric attack classification that simplifies the realization of Penetration Testing methodologies in terms of "Hacking Goals", "Hacking Tasks" and "Hacking Attacks". We use the goal-centric attack classification to realize a Web Attacker Knowledge Graph. We use common standard methodologies such as OWASP to define the "Hacking Tasks" of the Web Attacker Knowledge Graph. Future works will concentrate on extending the application domains and improving usability by providing an interface to interact with the knowledge graph database. Our second contribution is in the Offensive Security Automation's research field. We describe the realization of a platform that enables automating the Web Penetration Testing Activities. The "Executor Module" implements all the actions to attack the web target, while the "Orchestrator Module" embeds the attacker's behaviours that combine and orchestrate the attacks. We define a generic communication protocol that separates the attack execution from attack orchestration and allows the flexibility of new behavioural models in the platform. To assess the proposed platform, we show the integration of a behavioural model that discovers Reflected Cross-Site Scripting vulnerabilities. We show that it is possible to cover 27/41 web vulnerabilities through a mixed integration of Attack Tools and Attack Behaviours. The proposed architecture can be considered as the base for the realization of distributed attack systems. We have focused our scope on Web Application Penetration Testing. The system can be extended by implementing other types of Attack Models, as the architecture is flexible enough to allow easy implementation in different domains. Future works will be focused on the realization of behavioural models that leverage Artificial Intelligence and Natural Language Processing to automate any activity inside the current system. To test our platform, we need to design innovative solutions for virtualization environments. In the cyber range

field we give different contributions. As the first main contribution, we show how it is possible to overcome microservices-based limitations in reproducing vulnerable systems. As a second contribution, we propose a cloud model to deploy and orchestrate cyber ranges. In particular, we develop an AWS platform to demo-show the proposed architecture in a production-ready cloud environment. Future works in the cyber-range research area aim to improve our models in supported functionality and ease of implementation. First, we plan to evolve the architecture to dynamically deploy comprehensive Cyber Arenas other than the classical Cyber Ranges, providing Internet protocols emulation functionality. The Back-End Resource Manager will use orchestrators such as AWS Cloudformation to deploy Virtual and Hybrid Cyber Ranges. Moreover, we will harden the Cyber Range Environment perimeter by taking additional security measures such as setting up both Intrusion Detection and Intrusion Prevention systems. Another important work evolution can be related to the integration of relevant frameworks in security training. We could widen the scope of our platform by exploring the TIBER-EU framework [50], an important reference in the area of cyber-security training for the financial domain.

Contributions

The thesis work has been carried out also in the frame of collaborations with operational teams and/or industries. The cyber range has been evaluated and used by the Comando Interforze per le Operazioni Cibernetiche (CIOC) del Ministero della Difesa, whereas the automated pentesting platform, developed in the frame of a joint project with NTT Japan, has been included in the set of tools made available by NTT for its customers. During the PhD research work, three accepted papers has been produced, plus a forth submitted for publication; specifically:

- Hacking Goals: a goal-centric attack classification framework, published at the 32nd IEEE Testing Software and System conference (ICTSS 2020) [51]
- Discovering reflected Cross-Site Scripting vulnerabilities using a Multi-objective Reinforcement Learning environment, published on the Computer & Security Journal;
- On-demand deployment and orchestration of Cyber Ranges in the cloud, presented at the CEUR Workshop 2021 [28] ;

- Leveraging AI to optimize website structure discovery during Penetration Testing, currently under review at the Elsevier Big Data Research Journal, and available on Arxiv [52]

Bibliography

[1] Hessa Mohammed Zaher Al Shebli and Babak D Beheshti. A study on penetration testing process and tools. In *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–7. IEEE, 2018.

[2] Robert Shirey. Internet security glossary, version 2. 2007.

[3] Vinay M Igure and Ronald D Williams. Taxonomies of attacks and vulnerabilities in computer systems. *IEEE Communications Surveys & Tutorials*, 10(1):6–19, 2008.

[4] pentest.standard.org. *The Penetration Testing Execution Standard*. 2014.

[5] Mr Sean Barnum. Common attack pattern enumeration and classification (capec) schema. 2008.

[6] Igor Kotenko and Elena Doynikova. The capec based generator of attack scenarios for network security evaluation. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 436–441. IEEE, 2015.

[7] Jorge Lucangeli Obes, Carlos Sarraute, and Gerardo Richarte. Attack planning in the real world. *arXiv preprint arXiv:1306.4044*, 2013.

[8] Zoran Djuric. Waptt-web application penetration testing tool. *Advances in Electrical and Computer Engineering*, 14(1):93–102, 2014.

[9] Pulei Xiong and Liam Peyton. A model-driven penetration test framework for web applications. In *2010 Eighth International Conference on Privacy, Security and Trust*, pages 173–180. IEEE, 2010.

[10] Muhammad Saidu Aliero, Imran Ghani, Kashif Naseer Qureshi, and Mohd Foâad Rohani. An algorithm for detecting sql injection vulnerability using black-box testing. *Journal of Ambient Intelligence and Humanized Computing*, 11(1):249–266, 2020.

[11] H Winter. System security assessment using a cyber range. 2012.

[12] Muhammad Mudassar Yamin, Basel Katt, and Vasileios Gkioulos. Cyber ranges and security testbeds: Scenarios, functions, tools and architecture. *Computers & Security*, 88:101636, 2020.

[13] Diego Zamboni. *Detection of Intrusions and Malware, and Vulnerability Assessment (5 conf.)*. Springer, 2008.

[14] Arvind S Raj, Bithin Alangot, Seshagiri Prabhu, and Krishnashree Achuthan. Scalable and lightweight {CTF} infrastructures using application containers (pre-recorded presentation). In *2016 {USENIX} Workshop on Advances in Security Education ({ASE} 16)*, 2016.

[15] Yogesh Chandra and Pallaw Kumar Mishra. Design of cyber warfare testbed. In *Software Engineering*, pages 249–256. Springer, 2019.

[16] Razvan Beuran, Dat Tang, Cuong Pham, Ken-ichi Chinen, Yasuo Tan, and Yoichi Shinoda. Integrated framework for hands-on cybersecurity training: Cytrone. *Computers & Security*, 78:43–59, 2018.

[17] M. Karjalainen and T. Kokkonen. Comprehensive Cyber Arena; The Next Generation Cyber Range. *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 1–6, 2020.

[18] V. E. Urias, W. M. S. Stout, B. Van Leeuwen, and H. Lin. Cyber Range Infrastructure Limitations and Needs of Tomorrow: A Position Paper. *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5, 2018.

[19] Y. He, L. Yan, J. Liu, D. Bai, Z. Chen, X. Yu, D. Gao, and J. Zhu. Design of Information System Cyber Security Range Test System for Power Industry. *2019 IEEE Innovative Smart Grid Technologies - Asia (ISGT Asia)*, pages 1–5, 2019.

[20] G. Ãstby, K. N. Lovell, and B. Katt. Excon teams in cyber security training. *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1–6, 2020.

[21] M. Frank, M. Leitner, and T. Pahi. Design Considerations for Cyber Security Testbeds: A Case Study on a Cyber Security Testbed for Educations. *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 1–9, 2018.

[22] Dave Wichers. Owasp top-10 2013. *OWASP Foundation, February*, 2013.

[23] D. Stuttard and M. Pinto. *The web application hacker's handbook*. Wiley, 2013.

[24] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[25] CWE. Common weakness enumeration. `https://cwe.mitre.org/data/definitions/1026.html`.

[26] OWASP. Owasp testing guide, ver. 4.0. `https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP_Testing_Guide_v4.pdf`.

[27] Shaohua Wang, Ying Zou, Bipin Upadhyaya, and Joanna Ng. An intelligent framework for auto-filling web forms from different web applications. In *2013 IEEE Ninth World Congress on Services*, pages 175–179. IEEE, 2013.

[28] Francesco Caturano, Gaetano Perrone, and Simon Pietro Romano. Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment. *Computers & Security*, 103:102204, 2021.

[29] Ossama B Al-Khurafi and Mohammad A Al-Ahmad. Survey of web application vulnerability attacks. In *2015 4th International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 154–158. IEEE, 2015.

[30] Hasty Atashzar, Atefeh Torkaman, Marjan Bahrololum, and Mohammad H Tadayon. A survey on web application vulnerabilities and countermeasures. In *2011 6th International Conference on Computer Sci-*

ences and Convergence Information Technology (ICCIT), pages 647–652. IEEE, 2011.

[31] Phoebe Fasulo. 41 common web application vulnerabilities explained, Mar 2021.

[32] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[33] Amit K Nepal. Linux server & hardening security. 2014.

[34] MichaelStott. Michaelstott/crlf-injection-scanner: Command line tool for testing crlf injection on a list of domains.

[35] Vulnerability data base.

[36] Chenjj. Chenjj/corscanner: Fast cors misconfiguration vulnerabilities scanner.

[37] Secdec. Secdec/xssmap: Intelligent xss detection tool that uses human techniques for looking for reflected cross-site scripting (xss) vulnerabilities.

[38] D35m0nd142. D35m0nd142/lfisuite: Totally automatic lfi exploiter (+ reverse shell) and scanner.

[39] Nsedoc.

[40] 1N3. 1n3/massbleed: Massbleed ssl vulnerability scanner.

[41] Commixproject. Commixproject/commix: Automated all-in-one os command injection exploitation tool.

[42] Kurobeats. Kurobeats/fimap: Fimap is a little python tool which can find, prepare, audit, exploit and even google automatically for local and remote file inclusion bugs in webapps.

[43] Sqlmap.

[44] Http response splitting.

[45] Web security academy: Free online training from portswigger.

[46] Jeff Williams. Owasp testing guide, 2006.

[47] G. Perrone and S. P. Romano. The Docker Security Playground: A hands-on approach to the study of network security. *2017 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–8, 2017.

[48] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393. IEEE, 2015.

[49] Jan Vykopal, Radek Ošlejšek, Pavel Čeleda, Martin Vizvary, and Daniel Tovarňák. Kypo cyber range: Design and use cases. 2017.

[50] European Central Bank. Tiber-eu, Nov 2021.

[51] Francesco Caturano, Gaetano Perrone, and Simon Pietro Romano. Hacking goals: A goal-centric attack classification framework. In *IFIP International Conference on Testing Software and Systems*, pages 296–301. Springer, 2020.

[52] Diego Antonelli, Roberta Cascella, Gaetano Perrone, Simon Pietro Romano, and Antonio Schiano. Leveraging ai to optimize website structure discovery during penetration testing. *arXiv preprint arXiv:2101.07223*, 2021.